USENIX Association

Proceedings of the 12th USENIX Symposium on Networked Systems Design and Implementation (NSDI '15)

May 4–6, 2015
Oakland, CA, USA

Conference Organizers

Program Co-Chairs

Paul Barham, *Google*
Arvind Krishnamurthy, *University of Washington*

Program Committee

Atul Adya, *Google*
Aditya Akella, *University of Wisconsin–Madison*
Katerina Argyraki, *EPFL*
Hitesh Ballani, *Microsoft Research Cambridge*
Suman Banerjee, *University of Wisconsin–Madison*
Jeff Chase, *Duke University*
Jeff Dean, *Google*
Prabal Dutta, *University of Michigan*
Nick Feamster, *Georgia Institute of Technology*
Rodrigo Fonseca, *Brown University*
Deepak Ganesan, *University of Massachusetts Amherst*
Sharon Goldberg, *Boston University*
Andreas Haeberlen, *University of Pennsylvania*
Mark Handley, *University College London*
Jon Howell, *Microsoft Research Redmond*
Jaeyeon Jung, *Microsoft Research Redmond*
Sachin Katti, *Stanford University*
Ethan Katz-Bassett, *University of Southern California*
Kim Keeton, *HP Labs*
Changhoon Kim, *Barefoot Networks*
Eddie Kohler, *Harvard University*
Dejan Kostic, *KTH Royal Institute of Technology*
Kate Lin, *Academia Sinica Taiwan*
David Maltz, *Microsoft Redmond*
Petros Maniatis, *Google*
Alan Mislove, *Northeastern University*
Richard Mortier, *University of Nottingham*
Dushyanth Narayanan, *Microsoft Research Cambridge*
Venkat Padmanabhan, *Microsoft Research India*
KyoungSoo Park, *KAIST*
George Porter, *University of California, San Diego*
Rama Ramasubramanian, *Google*
Franzi Roesner, *University of Washington*
Srinivasan Seshan, *Carnegie Mellon University*
Neil Spring, *University of Maryland*
Lakshmi Subramanian, *New York University*
Kobus van der Merwe, *University of Utah*
Robbert van Renesse, *Cornell University*
Geoff Voelker, *University of California, San Diego*
Hakim Weatherspoon, *Cornell University*
Matei Zaharia, *MIT CSAIL and Databricks*
Nickolai Zeldovich, *MIT CSAIL*
Lidong Zhou, *Microsoft Research Asia*

Poster Session Co-Chairs

Rama Ramasubramanian, *Google*
Franzi Roesner, *University of Washington*

Steering Committee

Nick Feamster, *Georgia Institute of Technology*
Casey Henderson, *USENIX Association*
Arvind Krishnamurthy, *University of Washington*
Brian Noble, *University of Michigan*
Jennifer Rexford, *Princeton University*
Mike Schroeder
Alex C. Snoeren, *University of California, San Diego*

External Reviewers

Ranveer Chandra
Krishna Chintalapudi
David Chisnall
Advait Dixit
David Evans
Partha Kanuparthy
Harsha Madhyastha
Peter Peresini
Adrian Perrig
Thomas Ristenpart
Keith Winstein

NSDI ’15: 12th USENIX Symposium on Networked Systems Design and Implementation
May 4–6, 2015
Oakland, CA

Monday, May 4, 2015

Datacenters

SDN

Operational Systems Track 1

(Monday, May 4, continues on the next page)

Wireless

Tuesday, May 5, 2015

PHY Layer

Data Analytics

Operational Systems Track 2

Protocol Design and Implementation

Wednesday, May 6, 2015

Correctness

(Wednesday, May 6, continues on the next page)

Distributed Storage

Virtualization and Fault Tolerance

Message from the NSDI ’15 Program Co-Chairs

Welcome to NSDI ’15!

Over the years, NSDI has established itself as the top venue for work on networked and distributed systems. This year’s iteration is no exception, and we have an excellent program that spans the gamut from novel distributed systems to wireless technologies to data analytics.

We are particularly excited about the increasing popularity of the Operational Systems Track that is intended for a different breed of papers. Rather than pure research results, these papers describe experience with large-scale, deployed systems and networks. They offer a behind-the-scenes look at real networked and distributed systems, and insights which are otherwise hard to come by. This year’s session includes papers that span a broad range of topics, such as high-performance packet processing, scalable distributed systems, and services over the wide area.

We received 213 submissions (about the same as last year), and we accepted a record number of 42 papers. Our program committee had 45 members, including the co-chairs, with diverse expertise and experiences. The review process included two rounds of reviews, plenty of online discussion, and an in-person PC meeting. In the first round, each paper received three independent reviews. We advanced to the second round more than half of the submitted papers. In the second round, each paper received at least two additional reviews. Based on the reviews and online discussions, 81 papers were selected for discussion at the PC meeting. We sought external reviews sparingly, mostly in cases where the PC did not have sufficient expertise.

A conference like NSDI cannot succeed without the collective effort and support of many individuals and organizations. This effort starts with the authors, and we thank them for submitting the product of their hard work. Our PC members invested significant effort in the reviewing process, and we are grateful for their reviews, online discussions, and meeting participation. Special thanks to Geoff Voelker for managing papers with which both chairs were conflicted. We are also grateful to our external reviewers for lending their expertise, often on short notice. We thank Rama Ramasubramanian and Franzi Roesner for serving as Poster chairs. USENIX does a remarkable job of managing all non-technical aspects of the conference, and this year’s NSDI was no exception. Working with their staff, including Casey Henderson, Michele Nelson, and Julie Miller, was a pleasure. Sophie Ostlund at the University of Washington helped organize the PC meeting.

Finally, we thank you—the NSDI ’15 attendees. It is your participation and interest that sustains and nourishes the conference and our community.

Paul Barham, *Google*
Arvind Krishnamurthy, *University of Washington*
NSDI ’15 Program Co-Chairs

Queues don't matter when you can JUMP them!

Matthew P. Grosvenor Malte Schwarzkopf Ionel Gog Robert N. M. Watson
Andrew W. Moore Steven Hand[†] Jon Crowcroft

University of Cambridge Computer Laboratory

[†] now at Google, Inc.

Abstract

QJUMP is a simple and immediately deployable approach to controlling network interference in datacenter networks. Network interference occurs when congestion from throughput-intensive applications causes queueing that delays traffic from latency-sensitive applications. To mitigate network interference, QJUMP applies Internet QoS-inspired techniques to datacenter applications. Each application is assigned to a latency sensitivity level (or class). Packets from higher levels are rate-limited in the end host, but once allowed into the network can "jump-the-queue" over packets from lower levels. In settings with known node counts and link speeds, QJUMP can support service levels ranging from strictly bounded latency (but with low rate) through to line-rate throughput (but with high latency variance).

We have implemented QJUMP as a Linux Traffic Control module. We show that QJUMP achieves bounded latency and reduces in-network interference by up to 300×, outperforming Ethernet Flow Control (802.3x), ECN (WRED) and DCTCP. We also show that QJUMP improves average flow completion times, performing close to or better than DCTCP and pFabric.

1 Introduction

Many datacenter applications are sensitive to tail latencies. Even if as few as one machine in 10,000 is a straggler, up to 18% of requests can experience high latency [13]. This has a tangible impact on user engagement and thus potential revenue [8, 9].

One source of latency tails is *network interference*: congestion from throughput-intensive applications causes queueing that delays traffic from latency-sensitive applications. For example, Hadoop MapReduce can cause queueing that extends memcached request latency tails by 85 times the interference-free maximum (§2).

If memcached packets can somehow be prioritized to "jump-the-queue" over Hadoop's packets, memcached will no longer experience latency tails due to Hadoop. Of course, multiple instances of memcached may still interfere with *each other*, causing long queues or incast collapse [10]. If each memcached instance can be appropriately rate-limited at the origin, this too can be mitigated.

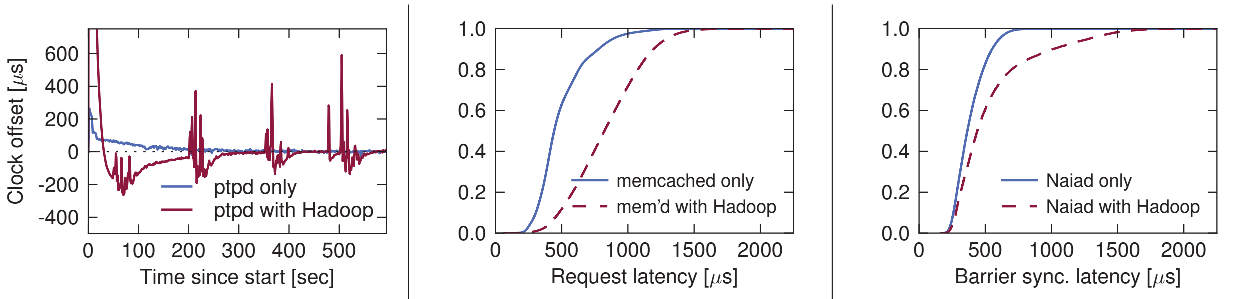
These observations are not new: QoS technologies like DiffServ [7] demonstrated that coarse-grained classification and rate-limiting can be used to control network latencies. Such schemes struggled for widespread deployment, and hence provided limited benefit [12]. However, unlike the Internet, datacenters have well-known network structures (i.e. host counts and link rates), and the bulk of the network is under the control of a single authority. In this environment, we can enforce system-wide policies, and calculate specific rate-limits which take into account worst-case behavior, ultimately allowing us to provide a guaranteed bound on network latency.

QJUMP is implemented via a simple rate-limiting Linux kernel module and application utility. QJUMP has four key features. It:

1. resolves network interference for latency-sensitive applications **without sacrificing utilization** for throughput-intensive applications;
2. offers **bounded latency** to applications requiring low-rate, latency-sensitive messaging (e.g. timing, consensus and network control systems);
3. is simple and **immediately deployable**, requiring no changes to hardware or application code; and
4. **performs close to or better** than competing systems, including ECN, 802.3x, DCTCP and pFabric, but is considerably less complex to understand, develop and deploy.

In this work, we consider only latency tails that result from in-network interference. Other work mitigates host-based sources of latency tails [14, 23, 30, 32, 36].

Please see *http://www.cl.cam.ac.uk/netos/qjump* for full details including the QJUMP source-code. In the electronic version of this paper, most of the figures and tables are clickable with links to a full experimental description and original datasets.



(a) Timeline of PTP synchronization offset. **(b)** CDF of memcached request latencies. **(c)** CDF of Naiad barrier sync. latencies.

Figure 1: Motivating experiments: Hadoop traffic interferes with *(a)* PTPd, *(b)* memcached and *(c)* Naiad traffic.

Setup	50th%	99th%
one host, idle network	85	126µs
two hosts, shared switch	110	130µs
shared source host, shared egress port	228	268µs
shared dest. host, shared ingress port	125	278µs
shared host, shared ingress and egress	221	229µs
two hosts, shared switch queue	**1,920**	**2,100µs**

Table 1: Median and 99th percentile latencies observed as `ping` and `iperf` share various parts of the network.

2 Motivation

We begin by showing that shared switch queues are the primary source of network interference. We then quantify the extent to which network interference impacts application-observable metrics of performance.

2.1 Where does the latency come from?

Network interference may occur at various places on the network path. Applications may share ingress or egress paths in the host, share the same network switch, or share the same queue in the same network switch. To assess the impact of interference in each of these situations, we emulate a latency-sensitive RPC application using `ping` and a throughput-intensive bulk transfer application by running two instances of `iperf`. Table 1 shows the results of arranging `ping` and `iperf` with various degrees of network sharing. Although any sharing situation results in interference, the effect is worst when applications share a congested switch queue.. In this case, the 99th percentile `ping` latency is degraded by over 16× compared to the unshared case.

2.2 How bad is it really?

Different applications use the network in different ways. To demonstrate the degree to which network interference affects different applications, we run three representative latency-sensitive applications (PTPd, memcached and Naiad) on a network shared with Hadoop (details in §6) and measure the effects.

1. Clock Synchronization Precise clock synchronization is important to distributed systems such as Google's Spanner [11]. PTPd offers microsecond-granularity time synchronization from a time server to machines on a local network. In Figure 1a, we show a timeline of PTPd synchronizing a host clock on both an idle network and when sharing the network with Hadoop. In the shared case, Hadoop's shuffle phases causes queueing, which delays PTPd's synchronization packets. This causes PTPd to temporarily fall 200–500µs out of synchronization; 50× worse than on an idle network.

2. Key-value Stores Memcached is a popular in-memory key-value store used by Facebook and others to store small objects for quick retrieval [25]. We benchmark memcached using the `memaslap` load generator[2] and measure the request latency. Figure 1b shows the distribution of request latencies on an idle network and a network shared with Hadoop. With Hadoop running, the 99th percentile request latency degrades by 1.5× from 779µs to 1196µs. Even worse, approximately 1 in 6,000 requests take over 200ms to complete[3], over 85× worse than the maximum latency seen on an idle network.

3. Iterative Data-Flow Naiad is a framework for distributed data-flow computation [24]. In iterative computations, Naiad's performance depends on low-latency state synchronization between worker nodes. To test Naiad's sensitivity to network interference, we execute a barrier synchronization benchmark (provided by the Naiad authors) with and without Hadoop running. Figure 1c shows the distribution of Naiad synchronization latencies in both situations. On an idle network, Naiad takes around 500µs at the 99th percentile to perform a four-way barrier synchronization. With interference, this grows to 1.1–1.5ms, a 2–3× performance degradation.

[2] *http://libmemcached.org*

[3] Likely because packet loss triggers the TCP minRTO timeout.

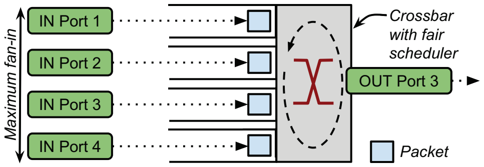


Figure 2: Packets fanning in to a four-port, virtual output queued switch. Output queues shown for port 3 only.

3 QJUMP System Design

Our exploratory experiments demonstrate that applications are sensitive to network interference, and that network interference occurs primarily as a result of shared switch queues. QJUMP therefore tackles network interference by reducing switch queueing: essentially, if we can reduce the amount of queueing in the network, then we will also reduce network interference. In the extreme case, if we can place a low, finite bound on queueing, then we can fully control network interference. This idea forms the basis of QJUMP.

In this section, we derive an intuitive model to place such a bound on queueing in any datacenter network topology. We first consider a single switch case, before extending the model to cover multiple switches. We then relax our model's throughput constraints and quantify the latency variance vs. throughput tradeoff. Finally, we describe how latency-sensitive traffic is allowed to "jump-the-queue" over high-throughput traffic.

Although the model we present is intuitive, it amounts to a simplification of the classic Parekh-Gallager theorem [27, 28]. The theorem shows that end-to-end delay can be bounded in a Weighted Fair Queueing (WFQ) network, provided that the network remains undersubscribed. In the Appendix, we show the relationship between our model and the theorem. In essence, we use the fact that datacenter networks have a well-known structure (unlike the Internet) to simplify the theorem, resulting in the version that we now present.

3.1 Bounded Queues – Bounded Latency

To begin, we assume an initially idle network in which each host is connected by a single link. We also assume that the link rate never decreases from the edge to the core of the network—an assumption that is true in any reasonable datacenter network.

Single Switch Queueing Model Consider the simplified model of a typical virtual-output queued (VOQ) layer 2 switch shown in Figure 2. The figure shows four input ports which are connected to four output ports via a crossbar. Only the output queues for port 3 are shown. One of two scenarios might occur at an instant in time: (*i*) only one input port sends packets to output port 3; or (*ii*) multiple input ports send packets to output port 3.

In the first case, a single sender can communicate with the destination port without queueing. Packets are only delayed by the processing delay across the switch, which is typically less than 0.5μs.

In the second case, packets arrive concurrently and only one packet can exit from the output port at a time. The switch scheduler must share access to this output by serializing the concurrent arrivals. In the worst case, the number of packets that arrive concurrently is equal to the *maximum fan-in* of the switch (see Figure 2), which is the number of input ports on the switch (four in this example). Thus, a packet may have to wait for up to *max fan-in* − 1 packets before it is serviced .

Multi Switch Queueing Model We can easily expand this understanding to cover multi-hop networks by treating the whole network as a single "big switch" (this is a version of the *hose-constraint* [15] model). Since we assume that each host has only one connection to the network, all packets "fanning in" to a host must eventually use this one link. This represents a mandatory serialization point, regardless of the core network topology. Given n hosts in the network, a packet may therefore experience at most *max network fan-in* $-1 = n-2 \approx n$ packets worth of delay. Knowing that a packet of size P will take $^P/_R$ seconds to transmit at link-rate R, we can therefore bound the maximum interference delay at:

$$\textit{worst case end-to-end delay} \leq n \times \frac{P}{R} + \varepsilon \qquad (1)$$

where n is the number of hosts, P the maximum packet size (in bits), R is the rate of the slowest link in bits per second and ε is the cumulative processing delay introduced by switch hops.

Network epochs So far, our model assumes that switch queues are initially empty and that the network is undersubscribed. In this case, Equation 1 offers an upper bound on end-to-end network delay. We refer to the result from Equation 1 as a *network epoch*. Intuitively, a network epoch is the maximum time that an idle network will take to service one packet from every sending host, regardless of the source, destination or timing of those packets. If all hosts are rate-limited so that they cannot issue more than one packet per epoch, no permanent queues can build up and the end-to-end network delay bound will be maintained forever.

One problem with a network epoch is that it is a global concept: to maintain it, all hosts must agree on when an epoch begins and when an epoch ends. This requires scheduling and precise timing. If all hosts in the network share a single time source, network epochs can be synchronized. Hardware time-stamped PTP synchronization on modern hardware can be used for micro-second granularity network scheduling [29]. PTP synchronization hardware is not yet ubiquitous. As an alternative, we can allow the network to become *mesochronous*. That is, we require all network epochs in the system to occur at the same *frequency*, but impose no restriction on the *phase* relationship between epochs. In this case, host-based timing is sufficient, so long as drift remains minimal over the sub-millisecond timespan of a network epoch.

This mesochronous relaxation does, however, affect our assumption of an initially idle network. A phase misalignment between hosts (or switches) means that a switch may encounter *two* packets within a host's network epoch: the first packet being issued at the end of an epoch and the second packet issued immediately at the start of the next epoch. The probability of this unfortunate alignment occurring decreases exponentially with scale. With as few as ten machines, the likelihood of waiting behind more than n packets is very small. Nevertheless, to ensure that the latency bound is *guaranteed*, we can accommodate the mesochronous case by doubling our worst-case latency bound. Our network epoch calculation thus becomes:

$$\textit{network epoch} = 2n \times \frac{P}{R} + \varepsilon \qquad (2)$$

This is a key property of QJUMP: if we rate-limit all hosts so that they can only issue one packet every network epoch, then no packet will take more than one network epoch to be delivered in the worst case.

3.2 Latency Variance vs. Throughput

Although the equation derived above provides an absolute upper bound on in-network delay, it also has the effect of aggressively restricting throughput. Formulating Equation 2 for throughput, we obtain:

$$\textit{throughput} = \frac{P}{\textit{network epoch}} \approx \frac{R}{2n} \qquad (3)$$

That is, as we increase the number of hosts n linearly, we decrease the throughput capacity for each host by a factor of $2n$. For example, with 1,000 hosts and a 10Gb/s edge we obtain an effective throughput of less than 5Mb/s per host. Clearly, this is not ideal.

We can improve this situation by making two observations. First, Equation 2 is pessimistic: it assumes that all hosts transmit to one destination at the worst time, which is unlikely given a realistic network and traffic distribution. Second, some applications (e.g. PTP) are more sensitive to interference than others (e.g. memcached, Naiad) whereas still other applications (e.g. Hadoop) are more sensitive to throughput restrictions.

From the first observation, we can relax the throughput constraints in Equation 2 by assuming that fewer than n hosts send to a single destination at the worst time. For example, if we assume that only 500 of the 1,000 hosts concurrently send to a single destination, then those 500 hosts can send at twice the rate and maintain the same network delay. More generally, we define a scaling factor f so that the assumed number of senders n' is given by:

$$n' = \frac{n}{f} \quad \text{where} \ \ 1 \leq f \leq n. \qquad (4)$$

Intuitively, f is a "throughput factor": as the value of f grows, so does the amount of bandwidth available.

From the second observation, some (but not all) applications can tolerate some degree of latency variance. For these applications, we aim for a statistical reduction in latency variance. This re-introduces a degree of statistical multiplexing to the network, but one that is more tightly controlled than in current networks. When the the value of f is too optimistic (i.e. the actual number of senders is greater than n'), some queueing may occur, resulting in network interference.

The probability that interference occurs increases with increasing values of f. At the upper bound ($f = n$), latency variance is no worse than in existing networks and full network throughput is available. At the lower bound ($f = 1$), latency is guaranteed, but with much reduced throughput. In essence, f quantifies the latency variance vs. throughput tradeoff.

3.3 Jump the Queue with Prioritization

We would like to use multiple values of f concurrently, so that different applications can benefit from the latency variance vs. throughput tradeoff that suits them best. To achieve this, we partition the network so that traffic from latency-sensitive applications (e.g. PTPd, memcached, Naiad) can "jump-the-queue" over traffic from throughput intensive applications (e.g. Hadoop).

Datacenter switches support the IEEE 802.1Q [18] standard which provides eight (0–7) hardware enforced "service classes" or "priorities". Priorities are rarely used in practice because priority selection can become a "race to the top". For example, memcached developers may assume that memcached traffic is the most important and should receive the highest priority. Meanwhile, Hadoop developers may assume that Hadoop traffic is the most important, and should similarly receive the highest priority. Since there is a limited number of priorities, neither can achieve an advantage and prioritization loses its value. QJUMP is different.

QJUMP couples priority values and rate-limits: for each priority, we assign a distinct value of f, with higher priorities receiving *smaller* values. Since a small value of f implies an aggressive rate limit, priorities become useful because they are no longer "free": QJUMP users must choose between low latency variance at low throughput (high priority) and high latency variance at high throughput (low priority).

We call the assignment of an f value to a priority a QJump *level*. The latency variance of a given QJump level is a function of the sum of the QJump levels above it. In Section 5, we discuss various ways of assigning f values to QJump levels.

4 QJump Implementation

QJump has two components: a rate-limiter to provide admission control to the network, and an application utility to configure unmodified applications to use QJump levels. In a multi-tenant environment, the rate-limiter is deployed as a component in the hypervisor and QJump is configured for the total number of virtual hosts. In a single-authority environment, the rate-limiter is deployed as an addition to the kernel network egress path and QJump is configured for the number of physical hosts.

Rate limiting QJump differs from many other systems that use rate-limiters. Instead of requiring a rate-limiter for each flow, each host only needs one coarse-grained rate-limiter per QJump level. This means that just eight rate-limiters per host are sufficient when using IEEE 802.1Q priorities. As a result, QJump rate-limiters can be implemented efficiently in software.

In our prototype, we use the queueing discipline (`qdisc`) mechanism offered by the Linux kernel traffic control (TC) subsystem to rate-limit packets. TC modules do not require kernel modifications and can be inserted and removed at runtime, making them flexible and easy to deploy. We also use Linux's built-in 802.1Q VLAN support to send layer 2 priority-tagged packets.

Listing 1 shows our a custom rate-limiter implementation. To keep the rate-limiter efficient, all operations quantify time in cycles. This requires us to initially convert the network epoch value from seconds into cycles (line 1). We then synthesize a clock from the CPU timestamp counter (`rdtsc`, line 6). This provides us with extremely fine-grained timing for the price of just one instruction on the critical path.

When a new packet arrives at the rate-limiter, it is classified into a QJump level using the priority tag found in its `sk_buff` (line 7). Users can set this priority directly in the application code, or assign priorities to unmodified binaries using our application utility. Next, the rate-limiter checks if a new epoch has begun. If so, it issues a fresh allocation of bytes to itself (lines 8–10). It then checks to see if sufficient bytes are remaining to send the packet in this network epoch (line 12). If so, the packet is forwarded to the driver (line 15–16), if not, the packet is dropped (line 13). In practice, packets are rarely dropped because our application utility also resizes socket buffers to apply early back-pressure.

Forwarded packets are mapped onto individual driver queues depending on the priority level. QJump therefore prioritizes low-latency traffic in the end-host itself, before packets are issued to the network card.

```
1  long epoch_cycles = to_cycles(network_epoch);
2  long timeout = start_time;
3  long bucket[NUM_QJUMP_LEVELS];
4
5  int qJumpRateLimiter(struct sk_buff* buffer) {
6    long cycles_now = asm("rdtsc"); /* read cycle ctr */
7    int level = buffer->priority;
8    if (cycles_now > timeout) { /* new token alloc? */
9      timeout += epoch_cycles;
10     bucket[level] = tokens[level];
11   }
12   if (buffer->len > bucket[level]) {
13     return DROP; /* tokens for epoch exhausted */
14   }
15   bucket[level] -= buffer->len;
16   sendToHWQueue(buffer, level);
17   return SENT;
18 }
```

Listing 1: QJump rate-limiter pseudocode.

Since Equation 2 assumes pessimal conditions, our rate-limiter also tolerates bursts up to the level-specific byte limit per epoch. This makes it compatible with hardware offload techniques such as TSO, LSO or GSO.

On our test machines, we found no measurable effect of the rate-limiter on CPU utilization or throughput. On average it imposes a cost of 35.2 cycles per packet ($\sigma = 18.6$; $99^{\text{th}}\%$ = 69 cycles) on the Linux kernel critical path of ≈8,000 cycles. This amounts to less than 0.5% overhead.

QJump Application Utility QJump requires that applications (or, specifically, sockets within applications) are assigned to QJump levels. This is easily done in application code directly with a `setsockopt()` using the `SO_PRIORITY` option. However, we would also like to support unmodified applications without recompilation. To achieve this, we have implemented a utility that dynamically intercepts socket setup system calls and alters their options. We inject the utility into unmodified executables via the Linux dynamic linker's `LD_PRELOAD` support (a similar technique to OpenOnload [31]).

The utility performs two tasks: (*i*) it configures socket priority values, and (*ii*) it sets socket send buffer sizes. Modifying socket buffer sizes is an optimization to apply early back-pressure to applications. If an application sends more data than its QJump level permits, an `ENOBUFS` error is returned rather than packets being dropped. While not strictly required, this optimization brings a significant performance benefit in practice as it helps avoid TCP retransmit timeouts (minRTOs).

5 Configuring QJump

A QJump deployment requires five parameters to be configured: (*i*) n, the number of hosts; (*ii*) P, the maximum packet size; (*iii*) R, the rate of the slowest edge link; (*iv*) ε, the edge-to-edge cumulative switch processing delay; and (*v*) f_i, the assumed fraction of concurrently transmitting hosts at each level.

Configuring R and ε As the topology of a datacenter network is static, the minimum link speed R and the cumulative switching delay ε do not vary. Typical values are $R = 10$Gb/s or 40Gb/s and ε = 1µs to 4µs.

Configuring P In §3.1, we defined P as the "maximum packet size". However, it is more correctly defined as the maximum number of bytes that can be issued into the network at the guaranteed latency level in a single network epoch. From Equation 2, the network epoch grows linearly with increasing P, so P should be kept small to keep the network epoch short. However, we also want P to be big enough to be useful. Benson *et al.* found that 30%–50% of packets in many datacenters contain fewer than 256 bytes [5]. This suggests that $\leq$256B packets are sufficient for some applications. For 1,000 hosts, setting P to 256 bytes results in a worst-case delay of $<$500µs.

Configuring n This usefulness of QJUMP depends on the size of the latency bound, which scales as a function of n. If all hosts in the network use QJUMP, then n can take a value of between 1,000 and 4,000 hosts and maintain a bound of 100-500µs using small messages of 64–256B. QJUMP can also be configured with n set as a subset of the hosts, provided that the remainder of hosts only use the lowest network priority.

Application-specific knowledge may also be exploited to increase the number of hosts that can participate in a QJUMP network. For example, a distribute/aggregate service may send requests to 10,000 hosts, but can be certain that fewer than 1,000 will respond. In this case, n can still be set to 1,000 hosts, but all 10,000 hosts can use QJUMP with guarantees. Finally, QJUMP scales with the network speed. On a faster network (e.g. a 40Gb/s edge), the same delay can be maintained for larger n (e.g. 16,000).

Configuring f_i The most complicated parameters to determine are the throughput factors f_i. Fortunately, each value of f_i is easily expressible as a rate-limit (e.g in Mb/s) which makes choosing values relatively intuitive (see §6 for examples). The best value for f_i depends on the desired latency distribution and the workload. The simplest configuration is to use only two QJUMP levels: (*i*) guaranteed latency (f_1 = 1) and (*ii*) maximum throughput (f_7 = n). Alternatively, a set of f_i values can be configured for a known application mix or for a known traffic distribution.

1. Known Application Mix Datacenter application mixes are often known, or information on application profiles can be obtained from users [4, 20, 21]. If application latency and throughput requirements can be estimated or measured, the QJUMP levels can be set to accommodate their needs.[4] In practice, simple benchmarks at different rate limits make it easy to characterize an application. We show an example in §6.5.

2. Known Traffic Distribution While the application mix in large datacenters can be complex, monitoring infrastructure supplies aggregate traffic statistics. An approximate distribution of flow sizes is often available [1, 5, 16]. For a known flow size distribution, f_i values can be configured to partition the traffic according to a desired latency variance vs. throughput distribution. We applied this method on a flow size CDF using a simple spreadsheet. This worked well in our experiments and simulations in §6.4.

6 Evaluation

We evaluate QJUMP both on a small deployment and in simulation. Our evaluation shows that QJUMP:

1. resolves network interference for a collection of real-world datacenter applications (§6.2);
2. outperforms Ethernet Flow Control (802.3x), ECN and DCTCP in our deployment (§6.3);
3. provides excellent flow completion times, close to or better than DCTCP [1] and pFabric [3] (§6.4);
4. is easily configurable, illustrated by examples of methods to determine QJUMP parameters (§6.5).

6.1 Experimental setup

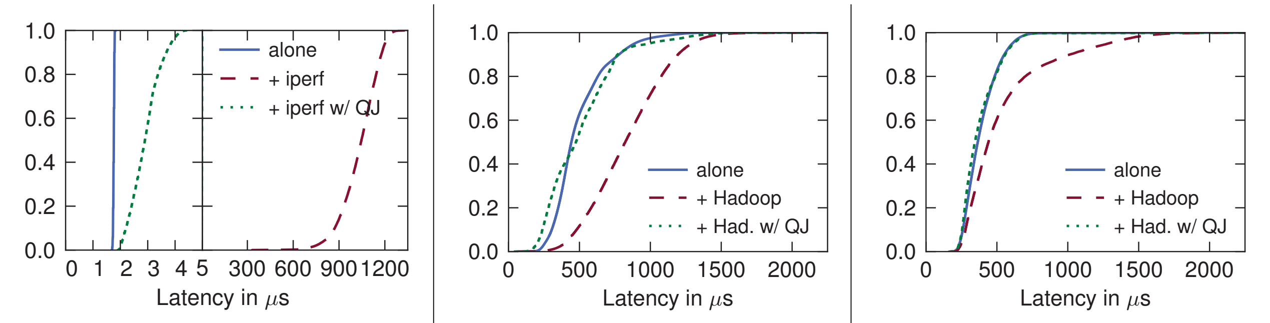
Our physical test-bed comprises an otherwise idle, 12 node cluster of recent AMD Opteron and Intel Xeon-based machines running Ubuntu 14.04 with Linux kernel 3.4.55. Each machine has one two-port 10Gb/s NIC installed. Our network is comprised of four Arista DCS-7124fx switches arranged as per Figure 4. We use `ptpd` v2.1.0 and memcached v1.4.14. We generate load for memcached using `memaslap` from `libmemcached` v1.0.15 running a binary protocol, mixed GET/SET workload of 1 KB requests in TCP mode with 128 concurrent requests. The Naiad experiments use v0.2.3 and the barrier-sync microbenchmark was supplied by the Naiad authors. Hadoop 2.0.0-mr1-cdh4.5.1 is deployed on eight of our twelve nodes, with the HDFS data in `tmpfs` and the replication factor set to six.[5] The Hadoop workload is a natural join between two uniformly randomly generated 512 MB data sets (39M rows each), which produces an output of 29 GB (1.5B rows).

6.2 QJUMP Resolves Network Interference

Our experiments in §2 showed that network interference degrades application performance. We now repeat those experiments with QJUMP enabled and show that QJUMP mitigates the network interference, resulting in near ideal

[4] There may be more applications than QJUMP levels. In this case, some levels will need to be shared between applications.

[5] This simulates the traffic a larger Hadoop cluster would generate.



(a) CDF of ping packet latency across a switch. Note the change in x-axis scale.

(b) QJUMP reduces memcached request latency: CDF of 9 million samples.

(c) QJ fixes Naiad barrier synchronization latency: CDF over 10k samples.

Figure 3: Application-level latency experiments: QJUMP (green, dotted line) mitigates the latency tails from Figure 1.

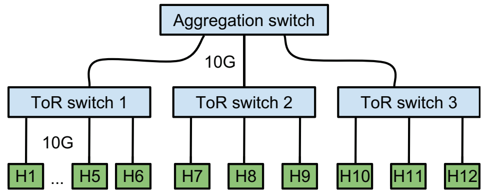


Figure 4: Network topology of our test-bed.

performance. We also show that in a realistic multi-application setting, QJUMP both resolves network interference and outperforms other readily available systems. We execute these experiments on the topology shown in Figure 4.

Low Latency RPC vs. Bulk Transfer Remote Procedure Calls (RPCs) and bulk data transfers represent extreme ends of the latency-bandwidth spectrum. QJUMP resolves network interference at these extremes. As in §2.1, we emulate RPCs and bulk data transfers using `ping` and `iperf` respectively. We measure in-network latency for the `ping` traffic directly using a high resolution Endace DAG capture card and two optical taps on either side of a switch. This verifies that queueing latency at switches is reduced by QJUMP. By setting `ping` to the highest QJUMP level ($f_7 = 1$), we reduce its packets' latency at the switch by over 300× (Figure 3a). The small difference between idle switch latency (1.6µs) and QJUMP latency (2–4µs) arises due a small on-chip FIFO through which the switch must process packets in-order. The switch processing delay, represented as ε in Equation 2, is thus no more than 4µs for each of our switches.

Memcached QJUMP resolves network interference experienced by memcached sharing a network with Hadoop. We show this by repeating the memcached experiments in §2.2. In this experiment, memcached is configured at an intermediate QJUMP level, rate-limited to 5Gb/s (above memcached's maximum throughput; see §6.5). Figure 3b shows the distribution (CDF) of memcached request latencies when running on an idle network, a shared network, and a shared network with QJUMP enabled. With QJUMP enabled, the request latencies are close to the ideal. The median latency improves from 824µs in the shared case to 476µs, a nearly 2× improvement.[6]

Naiad Barrier Synchronization QJUMP also resolves network interference experienced by Naiad [24], a distributed system for executing data parallel dataflow programs. Figure 3c shows the latency distribution of a four-way barrier synchronization in Naiad. On an idle network network, 90% of synchronizations take no more than 600µs. With interfering traffic from Hadoop, this value doubles to 1.2ms. When QJUMP is enabled, however, the distribution closely tracks the uncontended baseline distribution, despite sharing the network with Hadoop. QJUMP here offers a 2–5× improvement in application-level latency.

Multi-application Environment In real-world datacenters, a range of applications with different latency and bandwidth requirements share same infrastructure. QJUMP effectively resolves network interference in these shared, multi-application environments. We consider a datacenter setup with three different applications: `ptpd` for time synchronization, memcached for serving small objects and Hadoop for batch data analysis. Since resolving on-host interference is outside the scope of our work, we avoid sharing hosts between applications in these experiments and share only the network infrastructure.

Figure 5 *(top)* shows a timeline of average request latencies (over a 1ms window) for memcached and synchronization offsets for `ptpd`, each running alone on an otherwise idle network. Figure 5 *(middle)*, shows the two applications sharing the network with Hadoop. In this

[6] The distributions for the idle network and the QJUMP case do not completely agree due of randomness in the load generated.

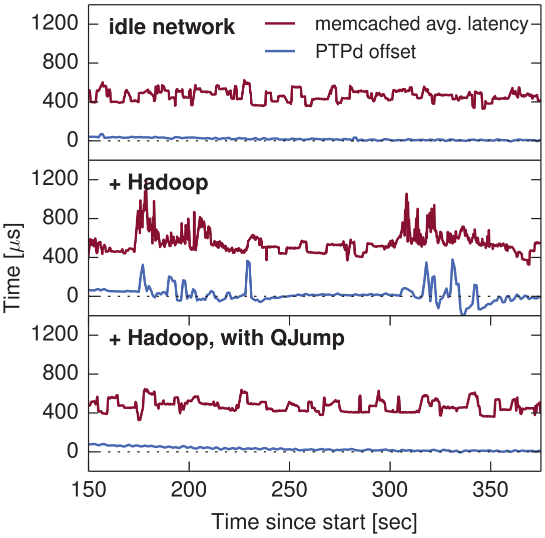


Figure 5: PTPd and memcached in isolation *(top)*, with interfering traffic from Hadoop *(middle)* and with the interference mitigated by QJUMP *(bottom)*.

case, average latencies increase for both applications and visible latency spikes (corresponding to Hadoop's shuffle phases) emerge. With QJUMP deployed, we assign ptpd to $f_7 = 1$, Hadoop to $f_0 = n = 12$ and memcached to $T_5 = 5$Gb/s $\implies$ $f_5 = 6$ (see §6.5). The three applications now co-exist without interference (Figure 5 *(bottom)*). Hadoop's performance is not noticeably affected by QJUMP, as we will further show in §6.3.

Distributed Atomic Commit One of QJUMP's unique features is its guaranteed latency level (described in §3.1). Bounded latency enables interesting new designs for datacenter coordination software such as SDN control planes, fast failure detection and distributed consensus systems. To demonstrate the usefulness of QJUMP's bounded latency level, we built a simple distributed two-phase atomic-commit (2PC) application.

The application communicates over TCP or over UDP with explicit acknowledgements and retransmissions. Since QJUMP offers reliable delivery, the coordinator can send its messages by UDP broadcast when QJUMP is enabled. This optimization yields a ≈30% throughput improvement over both TCP and UDP.

In Figure 6, we show the request rate for one coordinator and seven servers as a function of network interference. Interference is created with two traffic generators: on that generates a constant 10Gb/s of UDP traffic and another that sends fixed-size bursts followed by a 25ms pause. We report interference as the ratio of the burst size to the internal switch buffer size. Beyond a ratio of 200%, permanent queues build up in the switch. At this point the impact of retransmissions degrades throughput of the UDP and TCP implementations to 20% of the 10,000 requests/sec observed on an idle network. By contrast, the UDP-over-QJUMP implementation does not degrade as its messages "jump the queue". At high interference ratios (>200%), two-phase commit over QJUMP achieves 6.5× the throughput of standard TCP or UDP. Furthermore, QJUMP's reliable delivery and low latency enable very aggressive timeouts to be used for failure detection. Our 2PC system detects component failure within two network epochs (≈40µs on our network), far faster than typical failure detection timeouts (e.g. 150 ms in RAMCloud [26, §4.6]).

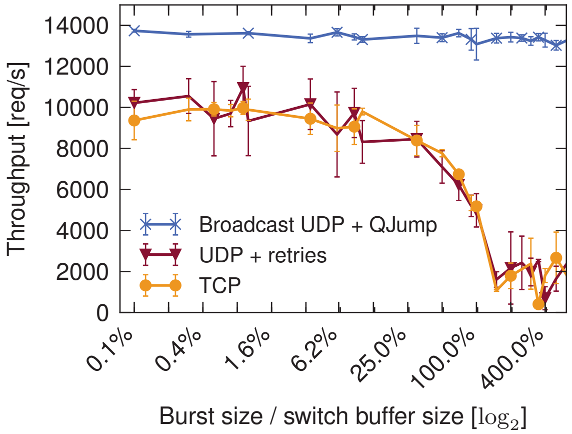


Figure 6: QJUMP offers constant two-phase commit throughput even at high levels of network interference.

6.3 QJUMP Outperforms Alternatives

Several readily deployable congestion control schemes exist, including Ethernet Flow Control (802.1x), Explicit Congestion Notifications (ECN) and Data Center TCP (DCTCP). We repeat the multi-application experiment described in §6.2 and show that QJUMP exhibits better interference control than other schemes.

Since interference is transient in these experiments, we measure the degree to which it affects applications using the root mean square (RMS) of each application-specific metric.[7] For Hadoop, PTPd and memcached, the metrics are job runtime, synchronization offset and request latency, respectively. Figure 7 shows six cases: an ideal case, a contended case and one for each of the four schemes used to mitigate network interference. All cases are normalized to the ideal case, which has each application running alone on an idle network. We discuss each result in turn.

Ethernet Flow Control Like QJUMP, Ethernet Flow Control is a data link layer congestion control mechanism. Hosts and switches issue special *pause* messages

[7] RMS is a statistical measure of the *magnitude* of a varying quantity [6, p. 64]. This is not the same as the root mean square error (RMSE), which measures prediction accuracy.

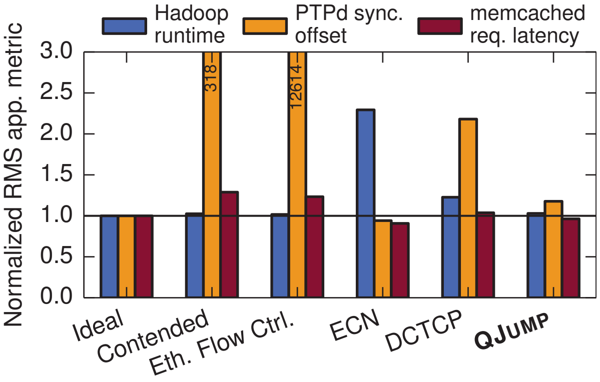


Figure 7: QJUMP comes closest to ideal performance for all of Hadoop, PTPd and memcached.

when their queues are nearly full, alerting senders to slow down. Figure 7 shows that Ethernet Flow Control has a limited positive influence on memcached, but increases the RMS for PTPd. Hadoop's performance remains unaffected.

Early Congestion Notification (ECN) ECN is a network layer mechanism in which switches indicate queueing to end hosts by marking TCP packets. Our Arista 7050 switch implements ECN with Weighted Random Early Detection (WRED). The effectiveness of WRED depends on an administrator correctly configuring upper and lower *marking thresholds*. We investigated ten different marking thresholds pairs, ranging between [5, 10] and [2560, 5120] ([upper, lower], in packets). None of these settings achieve ideal performance for all three applications, but the best compromise was [40, 80]. With this configuration, ECN very effectively resolves the interference experienced by PTPd and memcached. However, this comes at the expense of increased Hadoop runtimes.

Datacenter TCP (DCTCP) DCTCP uses the rate at which ECN markings are received to build an estimate of network congestion. It applies this to a new TCP congestion avoidance algorithm to achieve lower queueing delays [1]. We configured DCTCP with the recommended ECN marking thresholds of [65, 65]. Figure 7 shows that DCTCP reduces the variance in PTPd synchronization and memcached latency compared to the contended case. However, this comes at an increase in Hadoop job runtimes, as Hadoop's bulk data transfers are affected by DCTCP's congestion avoidance.

QJUMP Figure 7 shows that QJUMP achieves the best results. The variance in Hadoop, PTPd and memcached performance is close to (Hadoop, PTPd) or slightly better than (memcached) in the uncontended ideal case.

6.4 QJUMP Improves Flow Completion

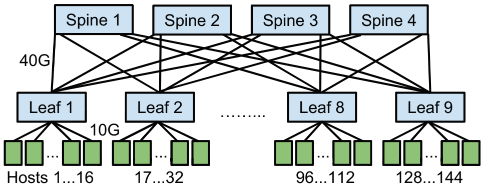


Figure 8: 144 node leaf-spine topology used for simulation experiments.

In addition to resolving network interference, QJUMP also provides excellent overall average and 99th percentile flow completion times (FCTs). Although QJUMP specifically optimizes tail latencies for small flows (at the expense of larger flows), doing so imposes a natural order on the network. This results in a surprisingly good overall network schedule with a generally positive impact on flow completion times.

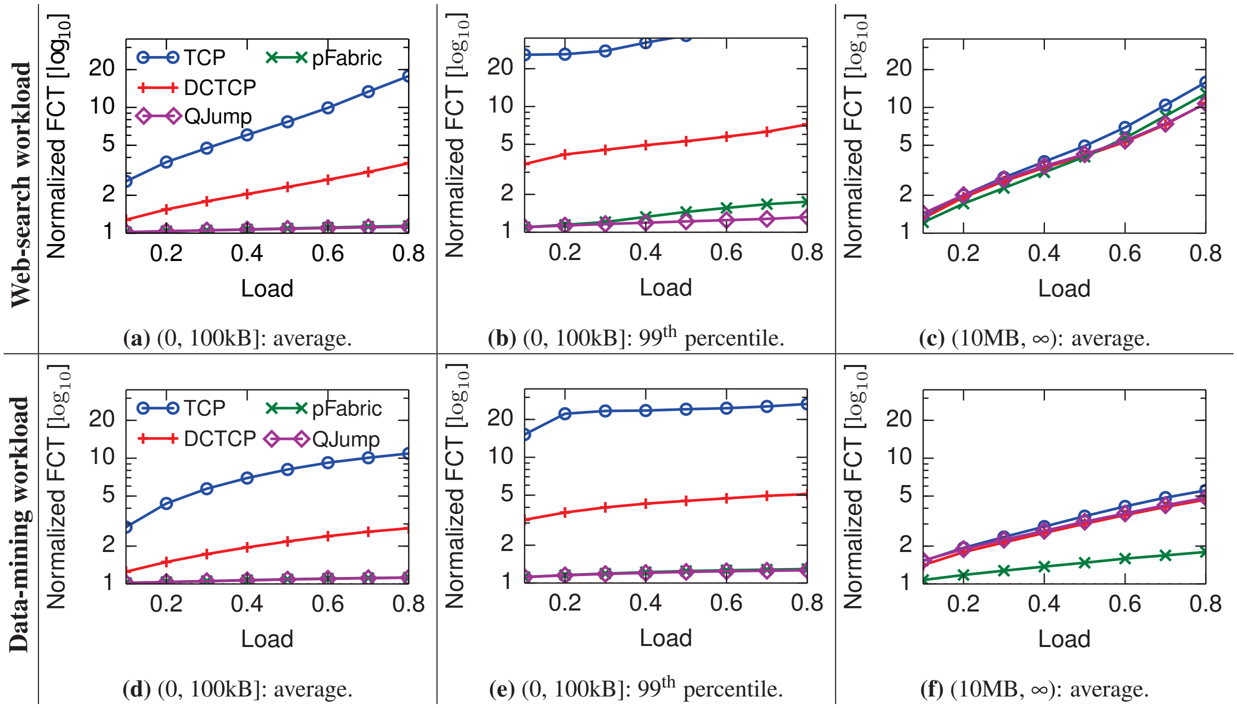
The pFabric architecture has been shown to schedule flows close to optimally [3]. Therefore, we compare QJUMP against pFabric to assess the quality of the network schedule it imposes. pFabric "is a clean-slate design [that] requires modifications both at the switches and the end-hosts" [3, §1] and is therefore only available in simulation. By contrast, QJUMP is far simpler and readily deployable, but applies rigid, global rate limits.

We compare QJUMP against a TCP baseline, DCTCP and pFabric by extending an `ns2` simulation provided by the authors of pFabric. This replicates the leaf-spine network topology used to evaluate pFabric (see Figure 8). We also run the same workloads derived from web search [1, §2.2] and data mining [16, §3.1] clusters in Microsoft datacenters, and show matching graphs in Figure 9.[8] As in pFabric, we normalize flows to their ideal flow completion time on an idle network.

Figure 9 reports the average and 99th percentile normalized FCTs for small flows (0kB, 100kB] and the average FCTs for large flows (10MB, ∞). For both workloads, QJUMP is configured with $P = 9$kB, $n = 144$, and $\{f_0 ... f_7\} = \{144, 100, 20, 10, 5, 3, 2, 1\}$. We chose this configuration based on the distribution of flow sizes in the web search workload. However, in practice it worked well for both workloads.

Despite its simplicity, QJUMP performs very well. As expected, it works best on short flows: on both workloads, QJUMP achieves average and 99th percentile FCTs close to or better than pFabric's. On the web-search workload, QJUMP beats pFabric by a margin of up to 32% at the 99th percentile (Fig. 9b). For larger flows, the results are mixed. On the web search workload, QJUMP

[8] An extended set of graphs for both of the workloads is available at *http://www.cl.cam.ac.uk/netos/qjump/sims.html*.



(a) (0, 100kB]: average. **(b)** (0, 100kB]: 99th percentile. **(c)** (10MB, ∞): average.

(d) (0, 100kB]: average. **(e)** (0, 100kB]: 99th percentile. **(f)** (10MB, ∞): average.

Figure 9: Normalized flow completion times in a 144-host simulation (1 is ideal): QJump outperforms TCP, DCTCP and pFabric for small flows. N.B.: log-scale *y*-axis; QJump and pFabric overlap in (a), (d) and (e).

outperforms pFabric by up to 20% at high load, but loses to pFabric by 15% at low load (Fig. 9c). On the data mining workload, QJump's average FCTs are between 30% and 63% worse than pFabric's (Fig. 9f).

In the data-mining workload, 85% of all *flows* transfer fewer than 100kB, but over 80% of the *bytes* are transferred in flows of greater than 100MB (less than 15% of the total flows). QJump's short epoch intervals cannot sense the difference between large flows, so it does not apply any rate-limiting (scheduling) to them. This results in sub-optimal behavior. A combined approach where QJump regulates interactions between large flows and small flows, while DCTCP regulates the interactions between different large flows might improve this.

6.5 QJump Configuration

As described in §5, QJump levels can be determined in several ways. One approach is to tune the levels to a specific mix of applications. For some applications, it is clear that they perform best at guaranteed latency (e.g. ptpd at $f_7 = 1$) or high rate (e.g. Hadoop at $f_0 = n$). For others, their performance at different throughput factors is less straightforward. Memcached is an example of such an application. It needs low request latency variance as well as reasonable request throughput. Figure 10 shows memcached's request throughput and latency as a function of rate-limiting. Peak throughput is reached at a rate allocation of around 5Gb/s. At the same point, the request latency also stabilizes. Hence, a rate-limit of 5Gb/s gives the best tradeoff for memcached. This point has the strongest interference control possible without throughput restrictions. To convert this to a throughput factor, we get $f_i = \frac{nT_i}{R}$ by rearranging Equation 2 for f_i. On our test-bed ($n = 12$ at R =10Gb/s), T_i =5Gb/s yields a throughput factor of $f = 6$. We can therefore choose a QJump level for memcached (e.g. f_4) and set it to a throughput factor ≥6.

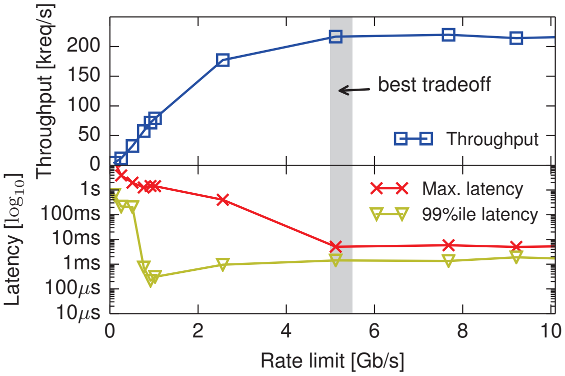


Figure 10: memcached throughput (top) and latency (bottom, $\log_{10}$) as a function of the QJump rate limit.

QJump offers a bounded latency level at throughput factor f_7. At this level, all packets admitted into the net-

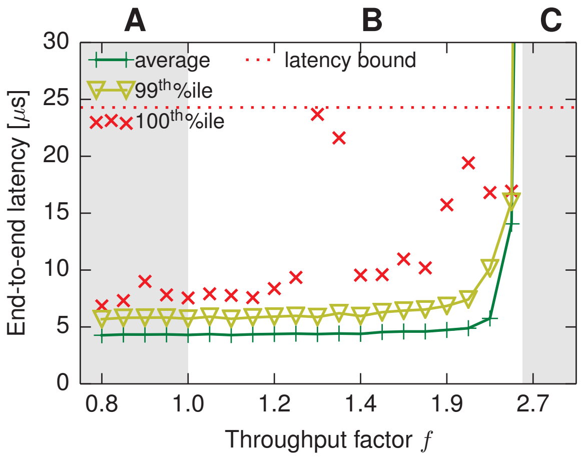


Figure 11: Latency bound validation: 60 host fan-in of f_7 and f_0 traffic; 100 million samples per data point.

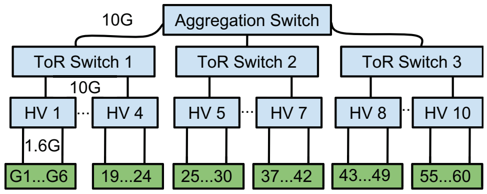


Figure 12: Latency bound validation topology: 10 hypervisors (HV) and 60 guests (G1..60) and 120 apps.

work must reach the destination by the end of the network epoch (§3.1). We now show that our model and the derived configuration perform correctly. To do this, we perform a scale-up emulation using a 60-host virtualized topology running on ten physical machines (see Figure 12). In this topology, each machine runs a "hypervisor" (Linux kernel) with a 10Gb/s uplink to the network. Each hypervisor runs six "guests" (processes) each with a 1.6Gb/s network connection. We provision QJUMP for the number of guests and run two applications on each guest: (*i*) a coordination service that generates one 256 byte packet per network epoch at the highest QJUMP level, and (*ii*) a bulk sender that issues 1500 byte packets as fast as possible at the lowest QJUMP level. All coordination requests are sent to a single destination.

Figure 11 shows the latency distribution of coordination packets as a function of the throughput factor at the highest QJUMP level, f_7. If the f_7 is set to less than 1.0 (region **A**), the latency bound is met (as we would expect). In region **B**, where f_7 is between 1.0 and 2.7, transient queueing affects some packets—as evident from the 100th percentile outliers—but all requests make it within the latency bound. Beyond $f_7 = 2.7$ (region **C**), permanent queueing occurs.

This experiment offers two further insights about QJUMP's rate-limiting: (*i*) at throughput factors near 1.0, the latency bound is usually still met, and (*ii*) via rate-limiting, QJUMP prevents latency-sensitive applications from interfering with their *own* traffic.

7 Related Work

Network congestion in datacenter networks is an active research area. Table 2 compares the properties of recent systems, including those we already compared against in §6.3 and §6.4. We categorize systems as *deployable* if they function on commodity hardware, unmodified transport protocols and unmodified application source code.

Fastpass [29] employs a global arbiter that times the admission of packets into the network and routes them. While Fastpass eliminates in-network queueing, requests for allocation must queue at the centralized arbiter.

EyeQ [22] primarily aims for bandwidth partitioning, although it also reduces latency tails. It, however, requires a full-bisection bandwidth network and a kernel patch in addition to a TC module.

Deadline Aware TCP (D^2TCP) [33] extends DCTCP's window adjustment algorithm with the notion of flow deadlines, scheduling flows with earlier deadlines first. Like DCTCP, D^2TCP requires switches supporting ECN;[9] it also requires inter-switch coordination, kernel and application modifications.

HULL combines DCTCP's congestion avoidance applied on network links' utilization (rather than queue length) with a special packet-pacing NIC [2]. Its rate-limiting is applied in reaction to ECN-marked packets.

D^3 [35] allocates bandwidth on a first-come-first-serve basis. It requires special switch and NIC hardware and modifies transport protocols.

PDQ uses Earliest Deadline First (EDF) scheduling to prioritize straggler flows, but requires coordination across switches and application changes.

DeTail [37] and pFabric [3] pre-emptively schedule flows using packet forwarding priorities in switches. DeTail also addresses load imbalance caused by poor flow hashing. Flow priorities are explicitly specified by modified applications (DeTail) or computed from the remaining flow duration (pFabric). However, both systems require special switch hardware: pFabric uses very short queues and 64-bit priority tags, and DeTail coordinates flows' rates via special "pause" and "unpause" messages.

SILO [21] employs a similar reasoning to QJUMP to estimate expected queue lengths. It places VMs according to traffic descriptions to limit queueing and paces hosts using "null" packets.

TDMA Ethernet [34] trades bandwidth for reduced queueing by time dividing network access, but requires invasive kernel changes and centralized coordination.

[9] Only one in five 10Gb/s switches we looked at supports ECN.

	System	Commodity hardware	Unmodified protocols	Unmodified OS kernel	Unmodified apps.	Coord. free	Flow deadlines	Bounded latency	Implemented
Deployable	Pause frames	✓	✓	✓	✓	✓	✗	✗	✓‡
	ECN	✓*, ECN	✓	✓	✓	✓	✗	✗	✓‡
	DCTCP [1]	✓*, ECN	✓*	✗	✓	✓	✗	✗	✓‡
	Fastpass [29]	✓	✓	✓, module	✓	✗	✗	✗	✓‡
	EyeQ [22]	✓*, ECN	✓	✗	✓	✗	✗	✗	✓‡
	QJUMP	✓	✓	✓, module	✓	✓	✓*	✓	✓‡
Not deployable	D^{2}TCP [33]	✓*, ECN	✓*	✗	✗	✗*	✓	✗	✓
	HULL [2]	✗	✓*	✗	✓	✓	✗	✗	✓*
	D^3 [35]	✗	✗	✗	✗	✓	✓	✗	✗*, softw.
	PDQ [17]	✗	✗	✗	✗	✗	✓	✗	✗
	pFabric [3]	✗	✗	✗	✓	✓	✓*	✗	✗
	DeTail [37]	✗	✓	✓	✗	✗*	✗	✗	✗*, softw.
	Silo [21]	✓	✓	✗	✓*	✗*	✓*, SLAs	✗	✓
	TDMA Eth. [34]	✓*	✓*	✗	✓*	✗	✗	✓	✓

Table 2: Comparison of related systems. *with caveats, see text; ‡implementation publicly available.

8 Discussion and Future Work

It would be ideal if applications were automatically classified into QJUMP levels. This requires overcoming a few challenges. First, the rate-limiter needs to be extended to calculate an estimate of instantaneous throughput for each application. Second, applications that exceed their throughput allocation must be moved to a lower QJUMP level, while applications that underutilize their allocation must be lifted to a higher QJUMP level. Third, some applications (e.g. Naiad) have latency-sensitive control traffic as well as throughput-intensive traffic that must be treated separately [19]. We leave this to future work.

9 Conclusion

QJUMP applies QoS-inspired concepts to datacenter applications to mitigate network interference. It offers multiple QJUMP levels with different latency variance vs. throughput tradeoffs, including bounded latency (at low rate) and full utilization (at high latency variance). In an extensive evaluation, we have demonstrated that QJUMP attains near-ideal performance for real applications and good flow completion times in simulations. Source code and data sets are available from *http://goo.gl/q1lpFC*.

Acknowledgements

We would like to thank Alex Ho and John Peach from Arista for arranging 10G switches for us. We would also like to thank Simon Peter, Srinivasan Keshav, Marwan Fayed, Rolf Neugebauer, Tim Harris, Antony Rowstron, Matthew Huxtable, Jeff Mogul and our anonymous reviewers for their valuable feedback. Thanks also go to our shepherd Jeff Dean. This work was supported by a Google Fellowship, EPSRC INTERNET Project EP/H040536/1, Defense Advanced Research Projects Agency (DARPA) and Air Force Research Laboratory (AFRL), under contract FA8750-11-C-0249. The views, opinions, and/or findings contained in this article are those of the authors and should not be interpreted as representing the official views or policies, either expressed or implied, of DARPA or the Department of Defense.

Appendix

The Parekh-Gallager theorem [27, 28] shows that Weighted Fair Queueing (WFQ) achieves a worst case delay bound given by the equation

$$end\ to\ end\ delay \leq \frac{\sigma}{g} + \sum_{i=1}^{K-1} \frac{P}{g_i} + \sum_{i=1}^{K} \frac{P}{r_i}, \qquad (5)$$

where all sources are governed by a leaky bucket abstraction with rate ρ and burst size σ, packets have a maximum size P and pass through K switches. For each switch i, there is a total rate r_i of which each connection (host) receives a rate g_i. g is the minimum of all g_i. It is assumed that $\rho \leq g$, i.e. the network is underutilized.

The final term in the equation adjusts for the difference between PGPS and GPS (Generalized Processor Sharing) for a non-idle network. Since we assume an idle network in our model (3.1), Equation 5 simplifies to

$$end\ to\ end\ delay \leq \frac{\sigma}{g} + \sum_{i=1}^{K-1} \frac{P}{g_i} \qquad (6)$$

If we assume that all hosts are given a fair share of the network—i.e. Fair Queueing rather than WFQ—then,

$$g_i = \frac{r_i}{n} \qquad (7)$$

where n is the number of hosts. Therefore the g (the minimum g_i) dominates. Since we assume an idle network, the remaining terms sum to zero. For a maximum burst size $\rho = P$, the equation therefore simplifies to

$$end\ to\ end\ delay \leq \frac{P}{g} = n \times \frac{P}{R} \qquad (8)$$

which is equivalent to the equation derived in Equation 1 (§3.1). The Parekh-Gallager theorem does not take into account the switch processing delay ε, since it is negligible compared to the end-to-end delay on the Internet.

References

[1] Alizadeh, M., Greenberg, A., Maltz, D. A., Padhye, J., Patel, P., Prabhakar, B., Sengupta, S., and Sridharan, M. Data center TCP (DCTCP). In *Proceedings of SIGCOMM* (2010), pp. 63–74.

[2] Alizadeh, M., Kabbani, A., Edsall, T., Prabhakar, B., Vahdat, A., and Yasuda, M. Less is more: trading a little bandwidth for ultra-low latency in the data center. In *Proceedings of NSDI* (2012), pp. 253–266.

[3] Alizadeh, M., Yang, S., Sharif, M., Katti, S., McKeown, N., Prabhakar, B., and Shenker, S. pFabric: Minimal Near-optimal Datacenter Transport. In *Proceedings of SIGCOMM* (2013), pp. 435–446.

[4] Ballani, H., Costa, P., Karagiannis, T., and Rowstron, A. Towards predictable datacenter networks. In *Proceedings of SIGCOMM* (2011), pp. 242–253.

[5] Benson, T., Akella, A., and Maltz, D. A. Network Traffic Characteristics of Data Centers in the Wild. In *Proceedings of IMC* (2010), pp. 267–280.

[6] Bissell, C., and Chapman, D. *Digital Signal Transmission*. Cambridge University Press, 1992.

[7] Blake, S., Black, D., Carlson, M., Davies, E., Wang, Z., and Weiss, W. RFC 2475: An architecture for differentiated services, Dec. 1998. Status: Proposed Standard.

[8] Brutlag, J. Speed Matters for Google Web Search. Tech. rep., Google. Available at: http://goo.gl/1qF8xt; accessed 24/09/2014.

[9] Card, S. K., Robertson, G. G., and Mackinlay, J. D. The information visualizer, an information workspace. In *Proceedings of CHI* (1991), pp. 181–186.

[10] Chen, Y., Griffith, R., Liu, J., Katz, R. H., and Joseph, A. D. Understanding TCP incast throughput collapse in datacenter networks. In *Proceedings of WREN* (2009), pp. 73–82.

[11] Corbett, J., Dean, J., Epstein, M., Fikes, A., Frost, C., Furman, J., Ghemawat, S., Gubarev, A., Heiser, C., et al. Spanner: Google's Globally-Distributed Database. In *Proceedings of OSDI* (2012), pp. 251–264.

[12] Crowcroft, J., Hand, S., Mortier, R., Roscoe, T., and Warfield, A. QoS's Downfall: At the bottom, or not at all! In *Proceedings of the ACM SIGCOMM Workshop on Revisiting IP QoS* (2003).

[13] Dean, J., and Barroso, L. A. The Tail at Scale: Managing Latency Variability in Large-Scale Online Services. *Commun. ACM 56*, 2 (Feb. 2013).

[14] Dobrescu, M., Argyraki, K., and Ratnasamy, S. Toward Predictable Performance in Software Packet-processing Platforms. In *Proceedings of NSDI* (2012), pp. 141–154.

[15] Duffield, N. G., Goyal, P., Greenberg, A., Mishra, P., Ramakrishnan, K. K., and van der Merwe, J. E. A Flexible Model for Resource Management in Virtual Private Networks. In *Proceedings of SIGCOMM* (1999), pp. 95–108.

[16] Greenberg, A., Hamilton, J. R., Jain, N., Kandula, S., Kim, C., Lahiri, P., Maltz, D. A., Patel, P., and Sengupta, S. VL2: a scalable and flexible data center network. In *Proceedings of SIGCOMM* (2009), pp. 51–62.

[17] Hong, C.-Y., Caesar, M., and Godfrey, P. B. Finishing flows quickly with preemptive scheduling. In *Proceedings of SIGCOMM* (2012), pp. 127–138.

[18] IEEE. Standard for local and metropolitan area networks, Virtual Bridged Local Area Networks. *IEEE Std. 802.11Q-2005* (2005).

[19] Isaacs, R. Tuning the performance of Naiad. Part 1: the network. Big Data at SVC blog, http://bit.ly/1gl5Cjk; accessed 25/09/2014.

[20] Jalaparti, V., Ballani, H., Costa, P., Karagiannis, T., and Rowstron, A. Bridging the tenant-provider gap in cloud services. In *Proceedings of SoCC* (2012), pp. 10:1–10:14.

[21] Jang, K., et al. Silo: Predictable Message Completion Time in the Cloud. Tech. rep., Microsoft Research, 2013. MSR-TR-2013-95.

[22] Jeyakumar, V., Alizadeh, M., Mazières, D., Prabhakar, B., Greenberg, A., and Kim, C. EyeQ: Practical Network Performance Isolation at the Edge. In *Proceedings of NSDI* (2013), pp. 297–311.

[23] Leverich, J., and Kozyrakis, C. Reconciling high server utilization and sub-millisecond quality-of-service. In *Proceedings of EuroSys* (2014), pp. 4:1–4:14.

[24] Murray, D. G., McSherry, F., Isaacs, R., Isard, M., Barham, P., and Abadi, M. Naiad: A Timely Dataflow System. In *Proceedings of SOSP* (2013), pp. 439–455.

[25] Nishtala, R., Fugal, H., Grimm, S., Kwiatkowski, M., Lee, H., Li, H. C., McElroy, R., Paleczny, M., Peek, D., Saab, P., Stafford, D., Tung, T., and Venkataramani, V. Scaling Memcache at Facebook. In *Proceedings of NSDI* (2013), pp. 385–398.

[26] Ongaro, D., Rumble, S. M., Stutsman, R., Ousterhout, J., and Rosenblum, M. Fast Crash Recovery in RAMCloud. In *Proceedings of SOSP* (2011), pp. 29–41.

[27] Parekh, A. K., and Gallager, R. G. A generalized processor sharing approach to flow control in integrated services networks: The single-node case. *IEEE/ACM Trans. Netw. 1*, 3 (June 1993), 344–357.

[28] Parekh, A. K., and Gallagher, R. G. A generalized processor sharing approach to flow control in integrated services networks: The multiple node case. *IEEE/ACM Trans. Netw. 2*, 2 (Apr. 1994), 137–150.

[29] Perry, J., Ousterhout, A., Balakrishnan, H., Shah, D., and Fugal, H. Fastpass: A centralized ”zero-queue” datacenter network. In *Proceedings of SIGCOMM* (2014), pp. 307–318.

[30] Peter, S., Li, J., Zhang, I., Ports, D. R. K., Woos, D., Krishnamurthy, A., Anderson, T., and Roscoe, T. Arrakis: The operating system is the control plane. In *Proceedings of OSDI 14* (Oct. 2014), pp. 1–16.

[31] Solarflare Communications Inc. www.openonload.org, November 2012.

[32] Tang, L., Mars, J., Vachharajani, N., Hundt, R., and Soffa, M.-L. The impact of memory subsystem resource sharing on datacenter applications. In *Proceedings of ISCA* (2011).

[33] Vamanan, B., Hasan, J., and Vijaykumar, T. Deadline-aware Datacenter TCP (D2TCP). *SIGCOMM Comput. Commun. Rev. 42*, 4 (Aug. 2012), 115–126.

[34] Vattikonda, B. C., Porter, G., Vahdat, A., and Snoeren, A. C. Practical TDMA for Datacenter Ethernet. In *Proceedings of Eurosys* (2012), pp. 225–238.

[35] Wilson, C., Ballani, H., Karagiannis, T., and Rowtron, A. Better never than late: Meeting deadlines in datacenter networks. In *Proceedings SIGCOMM* (2011), pp. 50–61.

[36] Xu, Y., Musgrave, Z., Noble, B., and Bailey, M. Bobtail: Avoiding long tails in the cloud. In *Proceedings of NSDI* (2013), pp. 329–342.

[37] Zats, D., Das, T., Mohan, P., Borthakur, D., and Katz, R. Detail: Reducing the flow completion time tail in datacenter networks. *SIGCOMM Comput. Commun. Rev. 42*, 4 (Aug. 2012), 139–150.

Explicit Path Control in Commodity Data Centers: Design and Applications

Shuihai Hu[1] Kai Chen[1] Haitao Wu[2] Wei Bai[1] Chang Lan[3]
Hao Wang[1] Hongze Zhao[4] Chuanxiong Guo[2]
[1]*SING Group @ Hong Kong University of Science and Technology*
[2]*Microsoft,* [3]*UC Berkeley,* [4]*Duke University*

Abstract

Many data center network (DCN) applications require explicit routing path control over the underlying topologies. This paper introduces XPath, a simple, practical and readily-deployable way to implement explicit path control, using existing commodity switches. At its core, XPath explicitly identifies an end-to-end path with a path ID and leverages a two-step compression algorithm to pre-install all the desired paths into IP TCAM tables of commodity switches. Our evaluation and implementation show that XPath scales to large DCNs and is readily-deployable. Furthermore, on our testbed, we integrate XPath into four applications to showcase its utility.

1 Introduction

Driven by modern Internet applications and cloud computing, data centers are being built around the world. To obtain high bandwidth and achieve fault tolerance, data center networks (DCNs) are often designed with multiple paths between any two nodes [3, 4, 13, 17, 18, 31]. Equal Cost Multi-Path routing (ECMP) [23] is the state-of-the-art for multi-path routing and load-balancing in DCNs [5, 17, 31].

In ECMP, a switch locally decides the next hop from multiple equal cost paths by calculating a hash value, typically from the source and destination IP addresses and transport port numbers. Applications therefore cannot explicitly control the routing path in DCNs.

However, many emerging DCN applications such as provisioned IOPS, fine-grained flow scheduling, bandwidth guarantee, etc. [5, 7, 8, 19, 21, 22, 25, 26, 39, 45], all require explicit routing path control over the underlying topologies (§2).

Many approaches such as source routing [36], MPLS [35], and OpenFlow [29] can enforce explicit path control. However, source routing is not supported in the hardware of the data center switches, which typically only support destination IP based routing. MPLS needs a signaling protocol, i.e., Label Distribution Protocol, to establish the path, which is typically used only for traffic engineering in core networks instead of application-level or flow-level path control. OpenFlow in theory can establish fine-grained routing paths by installing flow entries in the OpenFlow switches via the controller. But in practice, there are practical challenges such as limited flow table size and dynamic flow path setup that need to be addressed (see §6 for more details).

In order to address the scalability and deployment challenges faced by the above mentioned approaches, this paper presents XPath for flow-level explicit path control. XPath addresses the dynamic path setup challenge by giving a positive answer to the following question: can we pre-install all desired routing paths between any two nodes? Further, XPath shows that we can pre-install all these paths using the destination IP based forwarding TCAM tables of commodity switches[1].

One cannot enumerate all possible paths in a DCN as the number can be extremely large. However, we observe that DCNs (e.g., [2–4, 17, 18, 20]) do not intend to use *all* possible paths but a set of *desired* paths that are sufficient to exploit the topology redundancy (§2.2). Based on this observation, XPath focuses on pre-installing these desired paths in this paper. Even though, the challenge is that the sheer number of desired paths in large DCNs is still large, e.g., a Fattree ($k = 64$) has over 2^{32} paths among ToRs (Top-of-Rack switches), exceeding the size of IP table with 144K entries, by many magnitudes.

To tackle the above challenge, we introduce a two-step compression algorithm, i.e., paths to path sets aggregation and path ID assignment for prefix aggregation, which is capable of compressing a large number of paths to a practical number of routing entries for commodity switches (§3).

To show XPath's scalability, we evaluate it on various well-known DCNs (§3.3). Our results suggest that XPath effectively expresses tens of billions of paths using only tens of thousands of routing entries. For example, for Fattree(64), we pre-install 4 billion paths using ~64K entries[2]; for HyperX(4,16,100), we pre-install 17 billion paths using ~36K entries. With such algorithm, XPath

[1] The recent advances in switching chip technology make it ready to support 144K entries in IP LPM (Longest Prefix Match) tables of commodity switches (e.g., [1, 24]).

[2] The largest routing table size among all the switches.

easily pre-installs all desired paths into IP LPM tables with 144K entries, while still reserving space to accommodate more paths.

To demonstrate XPath's deployability, we implement it on both Windows and Linux platforms under the umbrella of SDN, and deploy it on a 3-layer Fattree testbed with 54 servers (§4). Our experience shows that XPath can be readily implemented with existing commodity switches. Through basic experiments, we show that XPath handles failure smoothly.

To showcase XPath's utility, we integrate it into four applications (from provisioned IOPS [25] to Map-reduce) to enable explicit path control and show that XPath directly benefits them (§5). For example, for provisioned IOPS application, we use XPath to arrange explicit path with necessary bandwidth to ensure the IOPS provisioned. For network update, we show that XPath easily assists networks to accomplish switch upgrades at zero traffic loss. For Map-reduce data shuffle, we use XPath to identify non-contention parallel paths in accord with the many-to-many shuffle pattern, reducing the shuffle time by over 3× compared to ECMP.

In a nutshell, the primary contribution of XPath is that it provides a practical, readily-deployable way to pre-install all the desired routing paths between any s-d pairs using existing commodity switches, so that applications only need to choose which path to use without worrying about how to set up the path, and/or the time cost or overhead of setting up the path.

To access XPath implementation scripts, please visit: `http://sing.cse.ust.hk/projects/XPath`.

The rest of the paper is organized as follows. §2 overviews XPath. §3 elaborates XPath algorithm and evaluates its scalability. §4 implements XPath and performs basic experiments. §5 integrates XPath into applications. §6 discusses the related work, and §7 concludes the paper.

2 Motivation and Overview

2.1 The need for explicit path control

Case #1: Provisioned IOPS: IOPS are input/output operations per second. Provisioned IOPS are designed to deliver predictable, high performance for I/O intensive workloads, such as database applications, that rely on consistent and fast response times. Amazon EBS provisioned IOPS storage was recently launched to ensure that disk resources are available whenever you need them regardless of other customer activity [25, 34]. In order to ensure provisioned IOPS, there is a need for necessary bandwidth over the network. Explicit path control is required for choosing an explicit path that can provide such necessary bandwidth (§5.1).

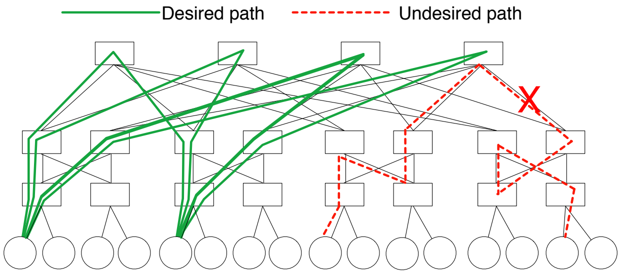


Figure 1: Example of the desired paths between two servers/ToRs in a 4-radix Fattree topology.

Case #2: Flow scheduling: Data center networks are built with multiple paths [4, 17]. To use such multiple paths, state-of-the-art forwarding in enterprise and data center environments uses ECMP to statically stripe flows across available paths using flow hashing. Because ECMP does not account for either current network utilization or flow size, it can waste over 50% of network bisection bandwidth [5]. Thus, to fully utilize network bisection bandwidth, we need to schedule elephant flows across parallel paths to avoid contention as in [5]. Explicit path control is required to enable such fine-grained flow scheduling, which benefits data intensive applications such as Map-reduce (§5.4).

Case #3: Virtual network embedding: In cloud computing, virtual data center (VDC) with bandwidth guarantees is an appealing model for cloud tenants due to its performance predictability in shared environments [7, 19, 45]. To accurately enforce such VDC abstraction over the physical topology with constrained bandwidth, one should be able to explicitly dictate which path to use in order to efficiently allocate and manage the bandwidth on each path (§5.3).

Besides the above applications, the need for explicit path control has permeated almost every corner of data center designs and applications, from traffic engineering (*e.g.*, [8, 22]), energy-efficiency (*e.g.*, [21]), to network virtualization (*e.g.*, [7, 19, 45]), and so on. In §5, we will study four of them.

2.2 XPath overview

To enable explicit path control for general DCNs, XPath explicitly identifies an end-to-end path with a path ID and attempts to pre-install all desired paths using IP LPM tables of commodity switches, so that DCN applications can use these pre-installed explicit paths easily without dynamically setting up them. In what follows, we first introduce what the desired paths are, and then overview the XPath framework.

Desired paths: XPath does not try to pre-install all possible paths in a DCN because this is impossible and impractical. We observe that when designing DCNs, operators do not intend to use all possible paths in the routing. Instead, they use a set of desired paths which are sufficient to exploit the topology redundancy. This is the case for many recent DCN designs such as [2–4, 17, 18, 20, 31]. For example, in a k-radix Fattree [4], they exploit $k^2/4$ parallel paths between any two ToRs for routing (see Fig. 1 for desired/undesired paths on a 4-radix Fattree); whereas in an n-layer BCube [18], they use $(n+1)$ parallel paths between any two servers. These sets of desired paths have already contained sufficient parallel paths between any s-d pairs to ensure good load-balancing and handle failures. As the first step, XPath focuses on pre-installing all these desired paths.

XPath framework: Fig. 2 overviews XPath. As many prior DCN designs [11, 17, 18, 31, 40], in our implementation, XPath employs a logically centralized controller, called *XPath manager*, to control the network. The XPath manager has three main modules: routing table computation, path ID resolution, and failure handling. Servers have client modules for path ID resolution and failure handling.

- *Routing table computation:* This module is the heart of XPath. The problem is how to compress a large number of desired paths (e.g., tens of billions) into IP LPM tables with 144K entries. To this end, we design a two-step compression algorithm: paths to path sets aggregation (in order to reduce unique path IDs) and ID assignment for prefix aggregation (in order to reduce IP prefix based routing entries). We elaborate the algorithm and evaluate its scalability in §3.
- *Path ID resolution:* In XPath, path IDs (in the format of 32-bit IP, or called routing IPs[3]) are used for routing to a destination, whereas the server has its own IP for applications. This entails path ID resolution which translates application IPs to path IDs. For this, the XPath manager maintains an IP-to-ID mapping table. Before a new communication, the source sends a request to the XPath manager resolving the path IDs to the destination based on its application IP. The manager may return multiple path IDs in response, providing multiple paths to the destination for the source to select. These path IDs will be cached locally for subsequent communications, but need to be forgotten periodically for failure handling. We elaborate this module and its implementation in §4.1.
- *Failure handling:* Upon a link failure, the detecting devices will inform the XPath manager. Then the XPath manager will in turn identify the affected paths and update the IP-to-ID table (i.e., disable the affected paths) to ensure that it will not return a failed path to a source that performs path ID resolution. The XPath source server handles failures by simply changing path IDs. This is because it has cached multiple path IDs for a destination, if one of them fails, it just uses a new live one instead. In the meanwhile, the source will request, from the manager, the updated path IDs to the destination. Similarly, upon a link recovery, the recovered paths will be added back to the IP-to-ID table accordingly. The source is able to use the recovered paths once the local cache expires and a new path ID resolution is performed.

  We note that XPath leverages failure detection and recovery outputs to handle failures. The detailed failure detection and recovery mechanisms are orthogonal to XPath, which focuses on explicit path control. In our implementation (§4.2), we adopt a simple TCP sequence based approach for proof-of-concept experiments, and we believe XPath can benefit from existing advanced failure detection and recovery literatures [15, 27].

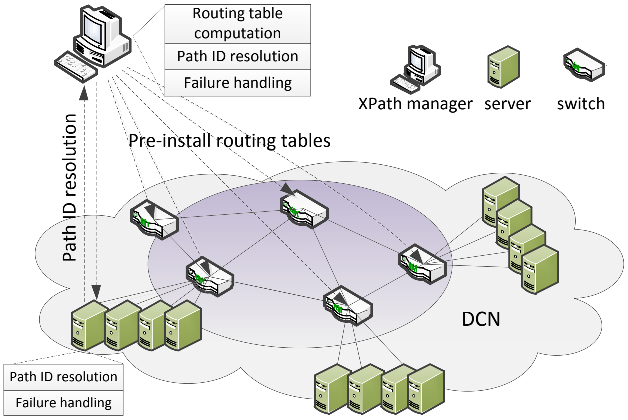


Figure 2: The XPath system framework.

Remarks: In this paper, XPath focuses on how to pre-install the desired paths, but it does not impose any constraint on how to use the pre-installed paths. On top of XPath, we can either let each server to select paths in a distributed manner, or employ an SDN controller to coordinate path selection between servers or ToRs in a centralized way (which we have taken in our implementation of this paper). In either case, the key benefit is that with XPath we do not need to dynamically modify the switches.

We also note that XPath is expressive and is able to pre-install all desired paths in large DCNs into commodity switches. Thus XPath's routing table recomputation is performed infrequently, and cases such as link failures or switch upgrade [26] are handled through changing path IDs rather than switch table reconfiguration. However, table recomputation is necessary for extreme cases like network wide expansion where the network topology has fundamentally changed.

[3] We use routing IPs and path IDs interchangeably in this paper.

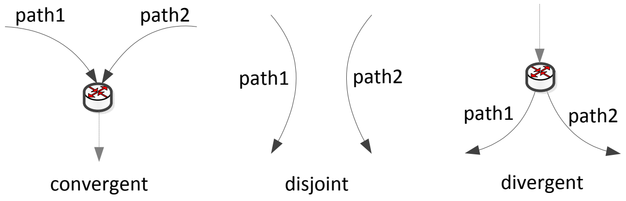


Figure 3: Three basic relations between two paths.

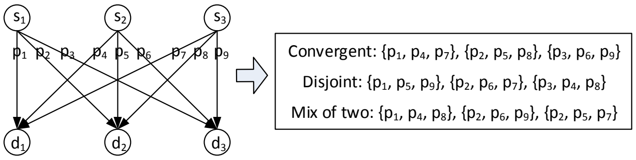

s1
s2
s3
p1 p2 p3 p4 p5 p6 p7 p8 p9
d1
d2
d3
Convergent: {p1, p4, p7}, {p2, p5, p8}, {p3, p6, p9}
Disjoint: {p1, p5, p9}, {p2, p6, p7}, {p3, p4, p8}
Mix of two: {p1, p4, p8}, {p2, p6, p9}, {p2, p5, p7}

Figure 4: Different ways of path aggregation.

Path set	Egress port	ID assignment (bad)	ID assignment (good)
$pathset_0$	0	0	4
$pathset_1$	1	1	0
$pathset_2$	2	2	2
$pathset_3$	0	3	5
$pathset_4$	1	4	1
$pathset_5$	2	5	3
$pathset_6$	0	6	6
$pathset_7$	0	7	7

Table 1: Path set ID assignment.

Path set	ID	Prefix	Egress port
$pathset_{1,4}$	$0,1$	$00*$	1
$pathset_{2,5}$	$2,3$	$01*$	2
$pathset_{0,3,6,7}$	$4,5,6,7$	$1**$	0

Table 2: Compressed table via ID prefix aggregation.

3 XPath Algorithm and Scalability

We elaborate the XPath two-step compression algorithm in §3.1 and §3.2. Then, we evaluate it on various large DCNs to show XPath's scalability in §3.3.

3.1 Paths to path sets aggregation (Step I)

The number of desired paths is large. For example, Fattree(64) has over 2^{32} paths between ToRs, more than what a 32-bit IP/ID can express. To reduce the number of unique IDs, we aggregate the paths that can share the same ID without causing routing ambiguity into a non-conflict path set, identified by a unique ID.

Then, what kinds of paths can be aggregated? Without loss of generality, two paths have three basic relations between each other, i.e., convergent, disjoint, and divergent as shown in Fig. 3. Convergent and disjoint paths can be aggregated using the same ID, while divergent paths cannot. The reason is straightforward: suppose two paths diverge from each other at a specific switch and they have the same ID $path1 = path2 = path\_id$, then there will be two entries in the routing table: $path\_id \rightarrow port_x$ and $path\_id \rightarrow port_y$, $(x \neq y)$. This clearly leads to ambiguity. Two paths can be aggregated without conflict if they do not cause any routing ambiguity on any switch when sharing the same ID.

Problem 1: Given the desired paths $P = \{p_1, \cdots, p_n\}$ of a DCN, aggregate the paths into non-conflict path sets so that the number of sets is minimized.

We find that the general problem of paths to non-conflict path sets aggregation is NP-hard since it can be reduced from the Graph vertex-coloring problem [41]. Thus, we resort to practical heuristics.

Based on the relations in Fig. 3, we can aggregate the convergent paths, the disjoint paths, or the mix into a non-conflict path set as shown in Fig. 4. Following this, we introduce two basic approaches for paths to path sets aggregation: convergent paths first approach (CPF) and disjoint paths first approach (DPF). The idea is simple. In CPF, we prefer to aggregate the convergent paths into the path set first until no more convergent path can be added in; Then we can add the disjoint paths, if exist, into the path set until no more paths can be added in. In DPF, we prefer to aggregate the disjoint paths into the path set first and add the convergent ones, if exist, at the end.

The obtained CPF or DPF path sets have their own benefits. For example, a CPF path set facilitates many-to-one communication for data aggregation because such an ID naturally defines a many-to-one communication channel. A DPF path set, on the other hand, identifies parallel paths between two groups of nodes, and such an ID identifies a many-to-many communication channel for data shuffle. In practice, users may have their own preferences to define customized path sets for different purposes as long as the path sets are free of routing ambiguity.

3.2 ID assignment for prefix aggregation (Step II)

While unique IDs can be much reduced through Step I, the absolute value is still large. For example, Fattree(64) has over 2 million IDs after Step I. We cannot allocate one entry per ID flatly with 144K entries. To address this problem, we further reduce routing entries using ID prefix aggregation. Since a DCN is usually under centralized control and the IDs of paths can be coordinately assigned, our goal of Step II is to assign IDs to paths in such a way that they can be better aggregated using prefixes in the switches.

3.2.1 Problem description

We assign IDs to paths that traverse the same egress port consecutively so that these IDs can be expressed using one entry via prefix aggregation. For example, in Table 1, 8 path sets go through a switch with 3 ports. A naïve (bad) assignment will lead to an uncompressable routing table with 7 entries. However, if we assign the paths that traverse the same egress port with consecutive IDs (good), we can obtain a compressed table with 3 entries as shown in Table 2.

To optimize for a single switch, we can easily achieve the optimal by grouping the path sets according to the egress ports and encoding them consecutively. In this way, the number of entries is equal to the number of ports. However, we optimize for all the switches simultaneously instead of one.

Problem 2. Let $T = \{t_1, t_2, \cdots, t_{|T|}\}$ be the path sets after solving Problem 1. Assigning (or ordering) the IDs for these path sets so that, after performing ID prefix aggregation, the largest number of routing entries among all switches is minimized.

In a switch, a block of consecutive IDs with the same egress port can be aggregated using one entry[4]. We call this an aggregateable ID block ($\mathbf{AIB}$). The number of such $\mathbf{AIB}$s indicates routing states in the switch[5]. Thus, we try to minimize the maximal number of $\mathbf{AIB}$s among all the switches through coordinated ID assignment.

To illustrate the problem, we use a matrix $\mathbf{M}$ to describe the relation between path sets and switches. Suppose switches have k ports (numbered as $1...k$), then we use $m_{ij} \in [0, k]$ $(1 \leq i \leq |S|, 1 \leq j \leq |T|)$ to indicate whether t_j goes through switch s_i, and if yes, which the egress port is. If $1 \leq m_{ij} \leq k$, it means t_j goes through s_i and the egress port is m_{ij}, and 0 otherwise means t_j does not appear on switch s_i.

$$\mathbf{M} = \begin{array}{c} \\ s_1 \\ s_2 \\ s_3 \\ \vdots \\ s_{|S|} \end{array} \begin{array}{c} \begin{array}{ccccc} t_1 & t_2 & t_3 & \cdots & t_{|T|} \end{array} \\ \begin{pmatrix} m_{11} & m_{12} & m_{13} & \cdots & m_{1|T|} \\ m_{21} & m_{22} & m_{23} & \cdots & m_{2|T|} \\ m_{31} & m_{32} & m_{33} & \cdots & m_{3|T|} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ m_{|S|1} & m_{|S|2} & m_{|S|3} & \cdots & m_{|S||T|} \end{pmatrix} \end{array}$$

To assign IDs to path sets, we use $\mathtt{f}(t_j) = r$ $(1 \leq r \leq |T|)$ to denote that, with an ID assignment $\mathtt{f}$, the ID for t_j is r (or ranks the r-th among all the IDs). With $\mathtt{f}$, we actually permutate columns on $\mathbf{M}$ to obtain $\mathbf{N}$. Column r in $\mathbf{N}$ corresponds to column t_j in $\mathbf{M}$, i.e., $[n_{1r}, n_{2r}, \cdots, n_{|S|r}]^{\mathbb{T}} = [m_{1j}, m_{2j}, \cdots, m_{|S|j}]^{\mathbb{T}}$.

$$\mathbf{N} = f(\mathbf{M}) = \begin{array}{c} \\ s_1 \\ s_2 \\ s_3 \\ \vdots \\ s_{|S|} \end{array} \begin{array}{c} \begin{array}{ccccc} 1 & 2 & 3 & \cdots & |T| \end{array} \\ \begin{pmatrix} n_{11} & n_{12} & n_{13} & \cdots & n_{1|T|} \\ n_{21} & n_{22} & n_{23} & \cdots & n_{2|T|} \\ n_{31} & n_{32} & n_{33} & \cdots & n_{3|T|} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ n_{|S|1} & n_{|S|2} & n_{|S|3} & \cdots & n_{|S||T|} \end{pmatrix} \end{array}$$

With matrix $\mathbf{N}$, we can calculate the number of $\mathbf{AIB}$s on each switch. To compute it on switch s_i, we only need to sequentially check all the elements on the i-th row. If there exist sequential non-zero elements that are the same, it means all these consecutive IDs share the same egress port and belong to a same $\mathbf{AIB}$. Otherwise, one more $\mathbf{AIB}$ is needed. Thus, the total number of $\mathbf{AIB}$s on switch s_i is:

$$\mathbf{AIB}(s_i) = 1 + \sum_{r=1}^{|T|-1} (n_{ir} \oplus n_{i(r+1)}) \qquad (1)$$

where $u \oplus v = 1$ if $u \neq v$ (0 is skipped), and 0 otherwise. With Equation 1, we can obtain the maximal number of $\mathbf{AIB}$s among all the switches: $\mathbf{MAIB} = \max_{1 \leq i \leq |S|} \{\mathbf{AIB}(s_i)\}$, and our goal is to find an $\mathtt{f}$ that minimizes $\mathbf{MAIB}$.

3.2.2 Solution

ID assignment algorithm: The above problem is NP-hard as it can be reduced from the 3-SAT problem [37]. Thus, we resort to heuristics. Our practical solution is guided by the following thought. Each switch s_i has its own local optimal assignment $\mathtt{f}_i$. But these individual local optimal assignments may conflict with each other by assigning different IDs to a same path set on different switches, causing an ID inconsistency on this path set. To generate a global optimized assignment $\mathtt{f}$ from the local optimal assignments $\mathtt{f}_i$s, we can first optimally assign IDs to path sets on each switch individually, and then resolve the ID inconsistency on each path set in an incremental manner. In other words, we require that each step of ID inconsistency correction introduces minimal increase on $\mathbf{MAIB}$.

Based on the above consideration, we introduce our ID_Assignment($\cdot$) in Algorithm 1. The main idea behind the algorithm is as follows.

- *First, we assign IDs to path sets on each switch individually.* We achieve the optimal result for each switch by simply assigning the path sets that have the same egress ports with consecutive IDs (lines 1–2).
- *Second, we correct inconsistent IDs of each path set incrementally.* After the first step, the IDs for a path

[4] The consecutiveness has local significance. Suppose path IDs 4, 6, 7 are on the switch (all exit through port p), but 5 are not present, then 4, 6, 7 are still consecutive and can be aggregated as $1{*}{*} \rightarrow p$.

[5] Note that the routing entries can be further optimized using subnetting and supernetting [16], in this paper, we just use $\mathbf{AIB}$s to indicate entries for simplicity, in practice the table size can be even smaller.

$$\mathbf{M} = \begin{array}{c|cccccc} & t_1 & t_2 & t_3 & t_4 & t_5 & t_6 \\ \hline s_1 & 1 & 1 & 1 & 2 & 1 & 2 \\ s_2 & 2 & 1 & 1 & 2 & 3 & 4 \\ s_3 & 1 & 2 & 2 & 2 & 3 & 2 \end{array} \rightarrow \mathbf{M'_0} = \begin{array}{c|cccccc} & t_1 & t_2 & t_3 & t_4 & t_5 & t_6 \\ \hline s_1 & 1(1) & 1(2) & 1(3) & 2(5) & 1(4) & 2(6) \\ s_2 & 2(3) & 1(1) & 1(2) & 2(4) & 3(5) & 4(6) \\ s_3 & 1(1) & 2(2) & 2(3) & 2(4) & 3(6) & 2(5) \end{array} \rightarrow \mathbf{M'_1} = \begin{array}{c|cccccc} & t_1 & t_2 & t_3 & t_4 & t_5 & t_6 \\ \hline s_1 & 1(3) & 1(2) & 1(1) & 2(5) & 1(4) & 2(6) \\ s_2 & 2(3) & 1(1) & 1(2) & 2(4) & 3(5) & 4(6) \\ s_3 & 1(3) & 2(2) & 2(1) & 2(4) & 3(6) & 2(5) \end{array} \rightarrow$$

$$\mathbf{M'_2} = \begin{array}{c|cccccc} & t_1 & t_2 & t_3 & t_4 & t_5 & t_6 \\ \hline s_1 & 1(3) & 1(2) & 1(1) & 2(5) & 1(4) & 2(6) \\ s_2 & 2(3) & 1(2) & 1(1) & 2(4) & 3(5) & 4(6) \\ s_3 & 1(3) & 2(2) & 2(1) & 2(4) & 3(6) & 2(5) \end{array} \rightarrow \ldots \rightarrow \mathbf{M'_6} = \begin{array}{c|cccccc} & t_1 & t_2 & t_3 & t_4 & t_5 & t_6 \\ \hline s_1 & 1(3) & 1(2) & 1(1) & 2(4) & 1(6) & 2(5) \\ s_2 & 2(3) & 1(2) & 1(1) & 2(4) & 3(6) & 4(5) \\ s_3 & 1(3) & 2(2) & 2(1) & 2(4) & 3(6) & 2(5) \end{array} \rightarrow \mathbf{N} = \begin{array}{c|cccccc} & 1 & 2 & 3 & 4 & 5 & 6 \\ \hline s_1 & 1 & 1 & 1 & 2 & 2 & 1 \\ s_2 & 1 & 1 & 2 & 2 & 4 & 3 \\ s_3 & 2 & 2 & 1 & 2 & 2 & 3 \end{array}$$

Figure 5: Walk-through example on Algorithm 1: for any element $x(y)$ in $\mathbf{M'_k}$ $(1 \le k \le 6)$, x is the egress port and y is the ID assigned to a path set t_j on switch s_i, red/green ys mean inconsistent/consistent IDs for path sets.

Algorithm 1 Coordinated ID assignment algorithm

```
   ID_Assignment($\mathbf{M}$)   /* $\mathbf{M}$ is initial matrix, $\mathbf{N}$ is output */;
1  foreach row $i$ of $\mathbf{M}$ (i.e., switch $s_i$) do
2      assign path sets $t_j (1 \le j \le |T|)$ having the same $m_{ij}$ values (i.e., egress ports) with consecutive IDs;
   /* path sets are optimally encoded on each switch locally, but one path set may have different IDs assigned with respect to different switches */;
3  $\mathbf{M'} \leftarrow \mathbf{M}$ with IDs specified for each $t_j$ in each $s_i$;
4  foreach column $j$ of $\mathbf{M'}$ (i.e., path set $t_j$) do
5      if $t_j$ has inconsistent IDs then
6          let $C = \{c_1, c_2, \cdots, c_k\}, (1 < k \le |S|)$ be the set of inconsistent IDs;
7          foreach $c \in C$ do
8              tentatively use $c$ to correct the inconsistency by swapping $c_i$ with $c$ on each relevant switch;
9              compute $\mathbf{MAIB}$;
10         ID($t_j$) $\leftarrow c$ with the minimal $\mathbf{MAIB}$;
11 return $\mathbf{N} \leftarrow \mathtt{f}(\mathbf{M'})$; /* $\mathbf{M'}$ is inconsistency-free */
```

set on different switches may be different. For any path set having inconsistent IDs, we resolve this as follows: we pick one ID out of all the inconsistent IDs of this path set and let other IDs be consistent with it provided that such correction leads to the minimal $\mathbf{MAIB}$ (lines 4–10). More specifically, in lines 6–9, we try each of the inconsistent IDs, calculate the associated $\mathbf{MAIB}$ if we correct the inconsistency with this ID, and finally pick the one that leads to the minimal $\mathbf{MAIB}$. The algorithm terminates after we resolve the ID inconsistencies for all the path sets.

In Fig. 5 we use a simple example to walk readers through the algorithm. Given $\mathbf{M}$ with 6 path sets across 3 switches, we first encode each switch optimally. This is achieved by assigning path sets having the same egress port with consecutive IDs. For example, on switch s_1, path sets t_1, t_2, t_3, t_5 exit from $port_1$ and t_4, t_6 from $port_2$, then we encode t_1, t_2, t_3, t_5 with IDs $1, 2, 3, 4$ and t_4, t_6 with $5, 6$ respectively. We repeat this on s_2 and s_3, and achieve $\mathbf{M'_0}$ with $\mathbf{MAIB} = 4$. However, we have inconsistent IDs (marked in red) for all path sets. For example, t_1 has different IDs $1, 3, 1$ on s_1, s_2, s_3 respectively. Then, we start to correct the inconsistency for each path set. For t_1 with inconsistent IDs $1, 3, 1$, we try to correct with IDs 1 and 3 respectively. To correct with ID 1, we exchange IDs 3 and 1 for t_1 and t_2 on switch s_2, and get $\mathbf{MAIB} = 5$. To correct with ID 3, we exchange IDs 1 and 3 for t_1 and t_3 on switch s_1 and s_3, and get $\mathbf{MAIB} = 4$. We thus choose to correct with ID 3 and achieve $\mathbf{M'_1}$ as it has minimal $\mathbf{MAIB} = 4$. We perform the same operation for the remaining path sets one by one and finally achieve $\mathbf{M'_6}$ with $\mathbf{MAIB} = 4$. Therefore, the final ID assignment is $\mathtt{f} : (t_1, t_2, t_3, t_4, t_5, t_6) \rightarrow (3, 2, 1, 4, 6, 5)$.

We note that the proposed algorithm is not optimal and has room to improve. However, it is effective in compressing the routing tables as we will show in our evaluation. One problem is the time cost as it works on a large matrix. We intentionally designed our Algorithm 1 to be of low time complexity, i.e., $O(|S|^2|T|)$ for the $|S| \times |T|$ matrix $\mathbf{M}$. Even though, we find that when the network scales to several thousands, it cannot return a result within 24 hours (see Table 4). Worse, it is possible that $|S| \sim 10^{4-5}$ and $|T| \sim 10^6$ or more for large DCNs. In such cases, even a linear time algorithm can be slow, not to mention any advanced algorithms.

Speedup with equivalence reduction: To speed up, we exploit DCN topology characteristics to reduce the runtime of our algorithm. The observation is that most DCN topologies are regular and many nodes are equivalent (or symmetric). These equivalent nodes are likely to have similar numbers of routing states for any given ID assignment, especially when the path sets are symmetrically distributed. The reason is that for two equivalent switches, if some path sets share a common egress port on one switch, most of these path sets, if not all, are likely to pass through a common egress port on another switch. As a result, no matter how the path sets are encoded, the ultimate routing entries on two equivalent switches tend to be similar. Thus, our hypothesis is that, by picking a representative node from each equivalence node class,

DCNs	Nodes #	Links #	**Original paths#**	**Max. entries # without compression**	**Path sets # after Step I compression**	**Max. entries # after Step I compression**	**Max. entries # after Step II compression**
Fattree(8)	208	384	**15,872**	**944**	**512**	**496**	**116**
Fattree(16)	1,344	3,072	**1,040,384**	**15,808**	**8,192**	**8,128**	**968**
Fattree(32)	9,472	24,576	**66,977,792**	**257,792**	**131,072**	**130,816**	**7,952**
Fattree(64)	70,656	196,608	**4,292,870,144**	**4,160,512**	**2,097,152**	**2,096,128**	**64,544**
BCube$(4, 2)$	112	192	**12,096**	**576**	**192**	**189**	**108**
BCube$(8, 2)$	704	1,536	**784,896**	**10,752**	**1,536**	**1,533**	**522**
BCube$(8, 3)$	6,144	16,384	**67,092,480**	**114,688**	**16,384**	**16,380**	**4,989**
BCube$(8, 4)$	53,248	163,840	**5,368,545,280**	**1,146,880**	**163,840**	**163,835**	**47,731**
VL2$(20, 8, 40)$	1,658	1,760	**31,200**	**6,900**	**800**	**780**	**310**
VL2$(40, 16, 60)$	9,796	10,240	**1,017,600**	**119,600**	**6,400**	**6,360**	**2,820**
VL2$(80, 64, 80)$	103,784	107,520	**130,969,600**	**4,030,400**	**102,400**	**102,320**	**49,640**
VL2$(100, 96, 100)$	242,546	249,600	**575,760,000**	**7,872,500**	**240,000**	**239,900**	**117,550**
HyperX$(3, 4, 40)$	2,624	2,848	**12,096**	**432**	**192**	**189**	**103**
HyperX$(3, 8, 60)$	31,232	36,096	**784,896**	**4,032**	**1,536**	**1,533**	**447**
HyperX$(4, 10, 80)$	810,000	980,000	**399,960,000**	**144,000**	**40,000**	**39,996**	**8,732**
HyperX$(4, 16, 100)$	6,619,136	8,519,680	**17,179,607,040**	**983,040**	**262,144**	**262,140**	**36,164**

Table 3: Results of XPath on the 4 well-known DCNs.

we can optimize the routing tables for all the nodes in the topology while spending much less time.

Based on the hypothesis, we improve the runtime of Algorithm 1 with equivalence reduction. This speedup makes no change to the basic procedure of Algorithm 1. Instead of directly working on $\mathbf{M}$ with $|S|$ rows, the key idea is to derive a smaller $\mathbf{M}^*$ with fewer rows from $\mathbf{M}$ using equivalence reduction, i.e., for all the equivalent nodes s_is in $\mathbf{M}$ we only pick one of them into $\mathbf{M}^*$, and then apply ID_Assignment$(\cdot)$ on $\mathbf{M}^*$. To this end, we first need to compute the equivalence classes among all the nodes, and there are many fast algorithms available for this purpose [10, 14, 28]. This improvement enables our algorithm to finish with much less time for various well-known DCNs while still maintaining good results as we will show subsequently.

3.3 Scalability evaluation

Evaluation setting: We evaluate XPath's scalability on 4 well-known DCNs: Fattree [4], VL2 [17], BCube [18], and HyperX [3]. Among these DCNs, BCube is a server-centric structure where servers act not only as end hosts but also relay nodes for each other. For the other 3 DCNs, switches are the only relay nodes and servers are connected to ToRs at last hop. For this reason, we consider the paths between servers in BCube and between ToRs in Fattree, VL2 and HyperX.

For each DCN, we vary the network size (Table 3). We consider $k^2/4$ paths between any two ToRs in Fattree(k), $(k+1)$ paths between any two servers in BCube(n, k), D_A paths between any two ToRs in VL2(D_A, D_I, T), and L paths between any two ToRs in HyperX(L, S, T)[6]. These paths do not enumerate all possible paths in the topology, however, they cover all desired paths sufficient to exploit topology redundancy in each DCN.

Our scalability experiments run on a Windows server with an Intel Xeon E7-4850 2.00GHz CPU and 256GB memory.

Main results: Table 3 shows the results of XPath algorithm on the 4 well-known DCNs, which demonstrates XPath's high scalability. Here, for paths to path sets aggregation we used CPF.

We find that XPath can effectively pre-install up to tens of billions of paths using tens of thousands of routing entries for very large DCNs. Specifically, for Fattree(64) we express 4 billion paths with 64K entries; for BCube(8,4) we express 5 billion paths with 47K entries; for VL2(100,96,100) we express 575 million paths with 117K entries; for HyperX(4,16,100) we express 17 billion paths with 36K entries. These results suggest that XPath can easily pre-install all desired paths into IP LPM tables with 144K entries, and in the meanwhile XPath is still able to accommodate more paths before reaching 144K.

Understanding the ID assignment: The most difficult part of the XPath compression algorithm is Step II (i.e., ID assignment), which eventually determines if XPath can pre-install all desired paths using 144K entries. The last two columns of Table 3 contrast the maximal entries before and after our coordinated ID assignment for each DCN.

We find that XPath's ID assignment algorithm can efficiently compress the routing entries by 2× to 32× for different DCNs. For example, before our coordinated ID assignment, there are over 2 million routing entries in the bottleneck switch (i.e., the switch with the largest routing table size) for Fattree(64), and after it, we achieve

[6] DCNs use different parameters to describe their topologies. In Fattree(k), k is the number of switch ports; in BCube(n, k), n is the number of switch ports and k is the BCube layers; in VL2(D_A, D_I, T), D_A/D_I are the numbers of aggregation/core switch ports and T is the number of servers per rack; in HyperX(L, S, T), L is the number of dimensions, S is the number of switches per dimension, and T is the number of servers per rack.

DCNs	Time cost (Second)	
	No equivalence reduction	Equivalence reduction
Fattree(16)	8191.121000	0.078000
Fattree(32)	>24 hours	4.696000
Fattree(64)	>24 hours	311.909000
BCube(8, 2)	365.769000	0.046000
BCube(8, 3)	>24 hours	6.568000
BCube(8, 4)	>24 hours	684.895000
VL2(40, 16, 60)	227.438000	0.047000
VL2(80, 64, 80)	>24 hours	3.645000
VL2(100, 96, 100)	>24 hours	28.258000
HyperX(3, 4, 40)	0.281000	0.000000
HyperX(4, 10, 80)	>24 hours	10.117000
HyperX(4, 16, 100)	>24 hours	442.379000

Table 4: Time cost of ID assignment algorithm with and without equivalence reduction for the 4 DCNs.

64K entries via prefix aggregation. In the worst case, we still compress the routing states from 240K to 117K in VL2(100,96,100). Furthermore, we note that the routing entries can be further compressed using traditional Internet IP prefix compression techniques, e.g., [16], as a post-processing step. Our ID assignment algorithm makes this prefix compression more efficient.

We note that our algorithm has different compression effects on different DCNs. As to the 4 largest topologies, we achieve a compression ratio of $\frac{2{,}096{,}128}{64{,}544}$=32.48 for Fattree(64), $\frac{262{,}140}{36{,}164} = 7.25$ for HyperX(4,16,100), $\frac{163{,}835}{47{,}731} = 3.43$ for BCube(8,4), and $\frac{239{,}900}{117{,}550} = 2.04$ for VL2(100,96,100) respectively. We believe one important decisive factor for the compression ratio is the density of the matrix $\mathbf{M}$. According to Equation 1, the number of routing entries is related to the non-zero elements in $\mathbf{M}$. The sparser the matrix, the more likely we achieve better results. For example, in Fattree(64), a typical path set traverses $\frac{1}{32}$ aggregation switches and $\frac{1}{1024}$ core switches, while in VL2(100,96,100), a typical path set traverses $\frac{1}{2}$ aggregation switches and $\frac{1}{50}$ core switches. This indicates that $\mathbf{M}_{\text{Fattree}}$ is much sparser than $\mathbf{M}_{\text{VL2}}$, which leads to the effect that the compression on Fattree is better than that on VL2.

Time cost: In Table 4, we show that equivalence reduction speeds up the runtime of the ID assignment algorithm. For example, without equivalence reduction, it cannot return an output within 24 hours when the network scales to a few thousands. With it, we can get results for all the 4 DCNs within a few minutes even when the network becomes very large. This is acceptable because it is one time pre-computation and we do not require routing table re-computation as long as the network topology does not change.

Effect of equivalence reduction: In Fig. 6, we compare the performance of our ID assignment with and without equivalence reduction. With equivalence reduction, we use $\mathbf{M}^*$ (i.e., part of $\mathbf{M}$) to perform ID assignment, and it

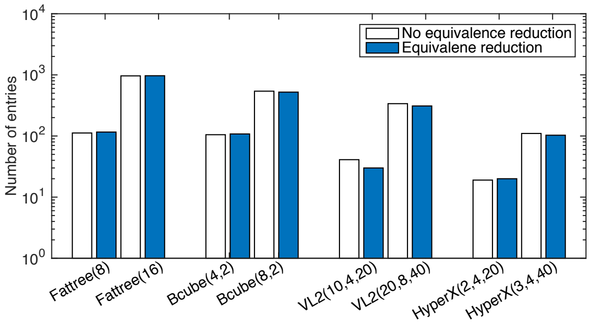


Figure 6: Effect of ID assignment algorithm with and without equivalence reduction for the 4 DCNs.

turns out that the results are similar to that without equivalence reduction. This partially validates our hypothesis in §3.2.2. Furthermore, we note that the algorithm with equivalence reduction can even slightly outperform that without it in some cases. This is not a surprising result since both algorithms are heuristic solutions to the original problem.

Results on randomized DCNs: We note that most other DCNs such as CamCube [2] and CiscoDCN [13] are regular and XPath can perform as efficiently as above. In recent work such as Jellyfish [39] and SWDC [38], the authors also discussed random graphs for DCN topologies. XPath's performance is indeed unpredictable for random graphs. But for all the Jellyfish topologies we tested, in the worst case, XPath still manages to compress over 1.8 billion paths with less than 120K entries. The runtime varies from tens of minutes to hours or more depending on the degree of symmetry of the random graph.

4 Implementation and Experiments

We have implemented XPath on both Windows and Linux platforms, and deployed it on a 54-server Fattree testbed with commodity switches for experiments. This paper describes the implementation on Windows. In what follows, we first introduce path ID resolution (§4.1) and failure handling (§4.2). Then, we present testbed setup and basic XPath experiments (§4.3).

4.1 Path ID resolution

As introduced in §2.2, path ID resolution addresses how to resolve the path IDs (i.e., routing IPs) for a destination. To achieve fault-tolerant path ID resolution, there are two issues to consider. First, how to distribute the path IDs of a destination to the source. The live paths to the destination may change, for example, due to link failures. Second, how to choose the path for a destination, and enforce such path selection in existing networks.

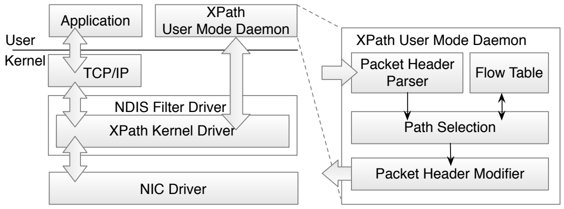


Figure 7: The software stacks of XPath on servers.

These two issues look similar to the name resolution in existing DNS. In practice, it is possible to return multiple IPs for a server, and balance the load by returning different IPs to the queries. However, integrating the path ID resolution of XPath into existing DNS may challenge the usage of IPs, as legacy applications (on socket communication) may use IPs to differentiate the servers instead of routing to them. Thus, in this paper, we develop a clean-slate XPath implementation on the XPath manager and end servers. Each server has its original name and IP address, and the routing IPs for path IDs are not related to DNS.

To enable path ID resolution, we implemented a XPath software module on the end server, and a module on the XPath manager. The end server XPath software queries the XPath manager to obtain the updated path IDs for a destination. The XPath manager returns the path IDs by indexing the IP-to-ID mapping table. From the path IDs in the query response, the source selects one for the current flow, and caches all (with a timeout) for subsequent communications.

To maintain the connectivity to legacy TCP/IP stacks, we design an IP-in-IP tunnel based implementation. The XPath software encapsulates the original IP packets within an IP tunnel: the path ID is used for the tunnel IP header and the original IP header is the inner one. After the tunnel packets are decapsulated, the inner IP packets are delivered to destinations so that multi-path routing by XPath is transparent to applications. Since path IDs in Fattree end at the last hop ToR, the decapsulation is performed there. The XPath software may switch tunnel IP header to change the paths in case of failures, while for applications the connection is not affected. Such IP-in-IP encapsulation also eases VM migration as VM can keep the original IP during migration.

We note that if VXLAN [42] or NVGRE [32] is introduced for tenant network virtualization, XPath IP header needs to be the outer IP header and we will need 3 IP headers which looks awkward. In the future, we may consider more efficient and consolidated packet format. For example, we may put path ID in the outer NVGRE IP header and the physical IP in NVGRE GRE Key field. Once the packet reaches the destination, the host OS then switches the physical IP and path ID.

In our implementation, the XPath software on end servers consists of two parts: a Windows Network Driver Interface Specification (NDIS) filter driver in kernel space and a XPath daemon in user space. The software stacks of XPath are shown in Fig. 7. The XPath filter driver is between the TCP/IP and the Network Interface Card (NIC) driver. We use the Windows filter driver to parse the incoming/outgoing packets, and to intercept the packets that XPath is interested in. The XPath user mode daemon is responsible for path selection and packet header modification. The function of the XPath filter driver is relatively fixed, while the algorithm module in the user space daemon simplifies debugging and future extensions.

In Fig. 7, we observe that the packets are transferred between the kernel and user space, which may degrade the performance. Therefore, we allocate a shared memory pool by the XPath driver. With this pool, the packets are not copied and both the driver and the daemon operate on the same shared buffer. We tested our XPath implementation (with tunnel) and did not observe any visible impact on TCP throughput at Gigabit line rate.

4.2 Failure handling

As introduced in §2.2, when a link fails, the devices on the failed link will notify the XPath manager. In our implementation, the communication channel for such notification is out-of-band. Such out-of-band control network and the controller are available in existing production DCNs [44].

The path IDs for a destination server are distributed using a query-response based model. After the XPath manager obtains the updated link status, it may remove the affected paths or add the recovered paths, and respond to any later query with the updated paths.

For proof-of-concept experiments, we implemented a failure detection method with TCP connections on the servers. In our XPath daemon, we check the TCP sequence numbers and switch the path ID once we detect that the TCP has retransmitted a data packet after a TCP timeout. The motivation is that the TCP connection is experiencing bad performance on the current path (either failed or seriously congested) and the XPath driver has other alternative paths ready for use. We note that this TCP based approach is sub-optimal and there are faster failure detection mechanisms such as BFD [15] or F10 [27] that can detect failures in 30μs, which XPath can leverage to perform fast rerouting (combining XPath with these advanced failure detection schemes is our future work). A key benefit of XPath is that it does not require route re-convergence and is loop-free during failure handling. This is because XPath pre-installs the backup paths and there is no need to do table re-computation unless all backup paths are down.

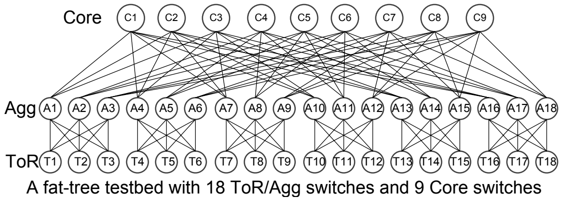


Figure 8: Fattree(6) testbed with 54 servers. Each ToR switch connects 3 servers (not drawn).

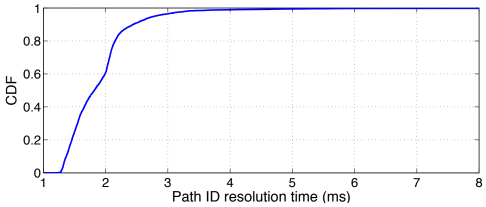


Figure 9: The CDF of path ID resolution time.

4.3 Testbed setup and basic experiments

Testbed setup: We built a testbed with 54 servers connected by a Fattree(6) network (as shown in Fig. 8) using commodity Pronto Broadcom 48-port Gigabit Ethernet switches. On the testbed, there are 18 ToR, 18 Agg, and 9 Core switches. Each switch has 6 GigE ports. We achieve these 45 virtual 6-port GigE switches by partitioning the physical switches. Each ToR connects 3 servers; and the OS of each server is Windows Server 2008 R2 Enterprise 64-bit version. We deployed XPath on this testbed for experimentation.

IP table configuration: On our testbed, we consider 2754 explicit paths between ToRs (25758 paths between end hosts). After running the two-step compression algorithm, the number of routing entries for the switch IP tables are as follows, ToR: 31∼33, Agg: 48, and Core: 6. Note that the Fattree topology is symmetric, the numbers of routing entries after our heuristic are almost the same for the switches at the same layer, which confirms our hypothesis in §3.2.2 that equivalent nodes are likely to have similar numbers of entries.

Path ID resolution time: We measure the path ID resolution time at the XPath daemon on end servers: from the time when the query message is generated to the time the response from the XPath manager is received. We repeat the experiment 4000 times and depict the CDF in Fig. 9. We observe that the 99-th percentile latency is 4ms. The path ID resolution is performed for the first packet to a destination server that is not found in the cache, or cache timeout. A further optimization is to perform path ID resolution in parallel with DNS queries.

XPath routing with and without failure: In this experiment, we show basic routing of XPath, with and without link failures. We establish 90 TCP connections from the 3 servers under ToR T1 to the 45 servers under ToRs T4 to T18. Each source server initiates 30 TCP connections in parallel, and each destination server hosts two TCP connections. The total link capacity from T1 is 3×1G=3G, shared by 90 TCP connections.

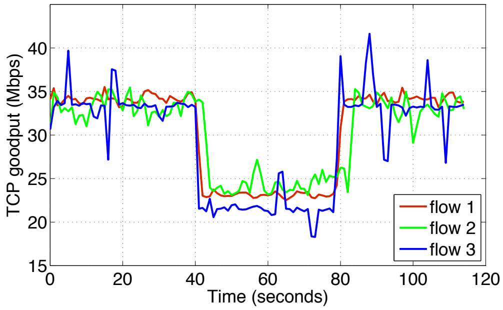


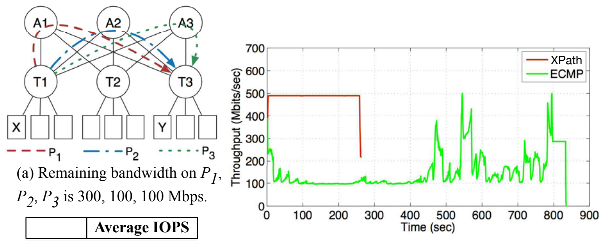
Figure 10: TCP goodput of three connections versus time on three phases: no failure, in failure, and recovered.

Given the 90 TCP connections randomly share 3 up links from T1, the load should be balanced overall. At around 40 seconds, we disconnect one link (T1 to A1). We use TCP sequence based method developed in §4.2 for automatic failure detection and recovery in this experiment. We then resume the link at time around 80 seconds to check whether the load is still balanced. We log the goodput (observed by the application) and show the results for three connections versus time in Fig. 10. Since we find that the throughput of all 90 TCP connections are very similar, we just show the throughput of one TCP connection for each source server.

We observe that all the TCP connections can share the links fairly with and without failure. When the link fails, the TCP connections traversing the failed link (T1 to A1) quickly migrate to the healthy links (T1 to A2 and A3). When the failed link recovers, it can be reused on a new path ID resolution after the timeout of the local cache. In our experiment, we set the cache timeout value as 1 second. However, one can change this parameter to achieve satisfactory recovery time for resumed links. We also run experiments for other traffic patterns, e.g., ToR-to-ToR and All-to-ToR, and link failures at different locations, and find that XPath works as expected in all cases.

5 XPath Applications

To showcase XPath's utility, we use it for explicit path support in four applications. The key is that, built on XPath, applications can freely choose which path to use without worrying about how to set up the path and the time cost or overhead of setting up the path. In this regard, XPath emerges as an interface for applications to use explicit paths conveniently, but does not make any choice on behalf of them.



(a) Remaining bandwidth on P_1, P_2, P_3 is 300, 100, 100 Mbps.

(b) Throughput and completion time of XPath and ECMP.

	Average IOPS
XPath	15274
ECMP	4547

(c) Average IOPS.

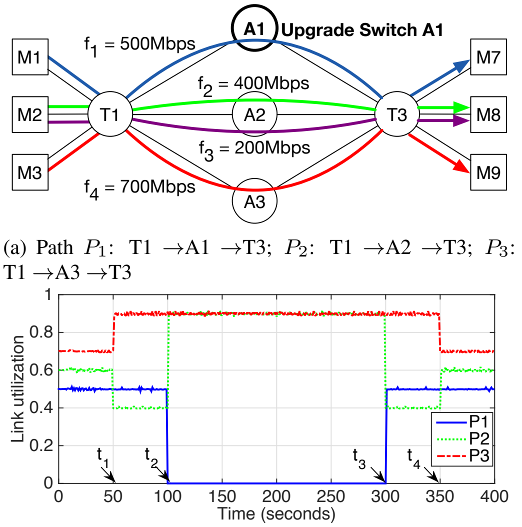
Figure 11: XPath utility case $\#1$: we leverage XPath to make necessary bandwidth easier to implement for provisioned IOPS.

5.1 XPath for provisioned IOPS

In cloud services, there is an increasing need for provisioned IOPS. For example, Amazon EBS enforces provisioned IOPS for instances to ensure that disk resources can be accessed with high and consistent I/O performance whenever you need them [25]. To enforce such provisioned IOPS, it should first provide necessary bandwidth for the instances [9]. In this experiment, we show XPath can be easily leveraged to use the explicit path with necessary bandwidth.

As shown in Fig. 11(a), we use background UDP flows to stature the ToR-Agg links and leave the remaining bandwidth on 3 paths (P_1, P_2 and P_3) between X-Y as 300Mpbs, 100Mbps, and 100Mbps respectively. Suppose there is a request for provisioned IOPS that requires 500Mbps necessary bandwidth (The provisioned IOPS is about 15000 and the chunk size is 4KB.). We now leverage XPath and ECMP to write 15GB data ($\approx$4 million chunks) through 30 flows from X to Y, and measure the achieved IOPS respectively. The storage we used for the experiment is Kingston V+200 120G SSD, and the I/O operations on the storage are sequential read and sequential write.

From Fig. 11(c), it can be seen that using ECMP we cannot provide the necessary bandwidth between X-Y for the provisioned IOPS although the physical capacity is there. Thus, the actual achieved IOPS is only 4547, and the write under ECMP takes much longer time than that under XPath as shown in Fig. 11(c). This is because ECMP performs random hashing and cannot specify the explicit path to use, hence it cannot accurately make use of the remaining bandwidth on each of the multiple paths for end-to-end bandwidth provisioning. In contrast, XPath can be easily leveraged to provide the required bandwidth due to its explicit path control. With XPath, we explicitly control how to use the three paths and accurately provide 500Mbps necessary bandwidth, achieving 15274 IOPS.



(a) Path P_1: T1 →A1 →T3; P_2: T1 →A2 →T3; P_3: T1 →A3 →T3

(b) Time t_1: move f_3 from P_2 to P_3; t_2: move f_1 from P_1 to P_2; t_3: move f_1 from P_2 to P_1; t_4: move f_3 from P_3 to P_2.

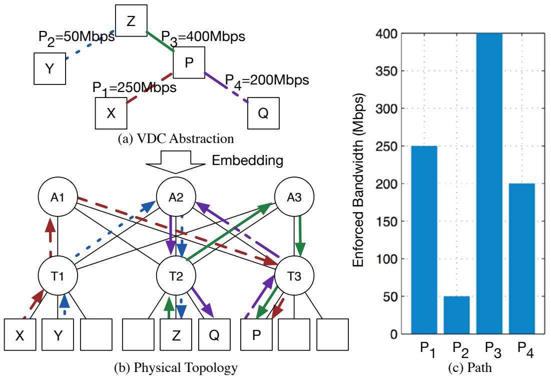
Figure 12: XPath utility case #2: we leverage XPath to assist zUpdate [26] to accomplish DCN update with zero loss.

5.2 XPath for network updating

In production data centers, DCN update occurs frequently [26]. It can be triggered by the operators, applications and various networking failures. zUpdate [26] is an application that aims to perform congestion-free network-wide traffic migration during DCN updates with zero loss and zero human effort. In order to achieve its goal, zUpdate requires explicit routing path control over the underlying DCNs. In this experiment, we show how XPath assists zUpdate to accomplish DCN update and use a switch firmware upgrade example to show how traffic migration is conducted with XPath.

In Fig. 12(a), initially we assume 4 flows (f_1, f_2, f_3 and f_4) on three paths (P_1, P_2 and P_3). Then we move f_1 away from switch A_1 to do a firmware upgrade for switch A_1. However, neither P_2 nor P_3 has enough spare bandwidth to accommodate f_1 at this point of time. Therefore we need to move f_3 from P_2 to P_3 in advance. Finally, after the completion of firmware upgrade, we move all the flows back to original paths. We leverage XPath to implement the whole movement.

In Fig. 12(b), we depict the link utilization dynamics. At time t_1, when f_3 is moved from P_2 to P_3, the link utilization of P_2 drops from 0.6 to 0.4 and the link utilization of P_3 increases from 0.7 to 0.9. At time t_2, when f_1 is moved from P_1 to P_2, the link utilization of P_1 drops from 0.5 to 0 and the link utilization of P_2 increases from 0.4 to 0.9. The figure also shows the changes of the link utilization at time t_3 and t_4 when moving f_3 back to P_2 and f_1 back to P_1. It is easy to see that with the help of XPath, P_1, P_2 and P_3 see no congestion and DCN update proceeds smoothly without loss.



(a) VDC Abstraction

(b) Physical Topology

(c) Path

Figure 13: XPath utility case $\#3$: we leverage XPath to accurately enforce VDC with bandwidth guarantees.

5.3 Virtual network enforcement with XPath

In cloud computing, virtual data center (VDC) abstraction with bandwidth guarantees is an appealing model due to its performance predictability in shared environments [7, 19, 45]. In this experiment, we show XPath can be applied to enforce virtual networks with bandwidth guarantees. We assume a simple SecondNet-based VDC model with 4 virtual links, and the bandwidth requirements on them are 50Mbps, 200Mbps, 250Mbps and 400Mbps respectively as shown in Fig. 13(a). We then leverage XPath's explicit path control to embed this VDC into the physical topology.

In Fig. 13(b), we show that XPath can easily be employed to use the explicit paths in the physical topology with enough bandwidth to embed the virtual links. In Fig. 13(c), we measure the actual bandwidth for each virtual link and show that the desired bandwidth is accurately enforced. However, we found that ECMP cannot be used to accurately enable this because ECMP cannot control paths explicitly.

5.4 Map-reduce data shuffle with XPath

In Map-reduce applications, many-to-many data shuffle between the map and reduce stages can be time-consuming. For example, Hadoop traces from Facebook show that, on average, transferring data between successive stages accounts for 33% of the running times of jobs [12]. Using XPath, we can explicitly express non-conflict parallel paths to speed up such many-to-many data shuffle. Usually, for a m-to-n data shuffle, we can use $(m{+}n)$ path IDs to express the communication patterns. The shuffle patterns can be predicted using existing techniques [33].

In this experiment, we selected 18 servers in two pods of the Fattree to emulate a 9-to-9 data shuffle by letting 9 servers in one pod send data to 9 servers in the other pod. We varied the data volume from 40G to over 400G. We compared XPath with ECMP.

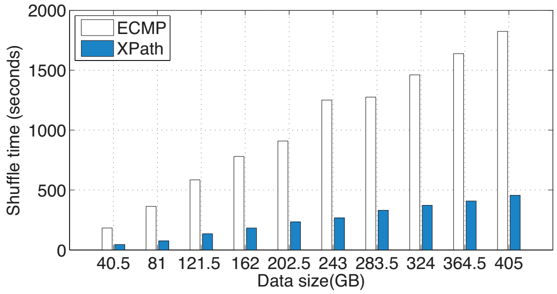


Figure 14: XPath utility case $\#4$: we leverage XPath to select non-conflict paths to speed up many-to-many data shuffle.

In Fig. 14, it can be seen that by using XPath for data shuffle, we can perform considerably better than randomized ECMP hash-based routing. More specifically, it reduces the shuffle time by over $3\times$ for most of the experiments. The reason is that XPath's explicit path IDs can be easily leveraged to arrange non-interfering paths for shuffling, thus the network bisection bandwidth is fully utilized for speedup.

6 Related Work

The key to XPath is explicit path control. We note that many other approaches such as source routing [36], MPLS [35], OpenFlow [29] and the like, can also enable explicit path control. However, each of them has its own limitation.

OpenFlow [29] has been used in many recent proposals (e.g., [5, 8, 21, 22, 26]) to enable explicit path control. OpenFlow can establish fine-grained explicit routing path by installing flow entries in the switches via the OpenFlow controller. But in current practice, there are still challenges such as small flow table size and dynamic flow entries setup that need to be solved. For example, the on-chip OpenFlow forwarding rules in commodity switches are limited to a small number, typically 1–4K. To handle this limitation, recent solutions, e.g. [22], dynamically change, based on traffic demand, the set of live paths available in the network at different times through dynamic flow table configurations, which could potentially introduce non-trivial implementation overhead and performance degradation. XPath addresses such challenge by pre-installing all desired paths into IP LPM tables. In this sense, XPath complements existing OpenFlow-based solutions in terms of explicit path control, and in the meanwhile, the OpenFlow framework may still be able to be used as a way for XPath to pre-configure the switches and handle failures.

Source routing is usually implemented in software and slow paths, and not supported in the hardware of the data center switches, which typically only support destination IP based routing. Compared to source routing, XPath is readily deployable without waiting for new hardware capability; and XPath's header length is fixed while it is variable for source routing with different path lengths.

With MPLS, paths can also be explicitly set up before data transmission using MPLS labels. However, XPath is different from MPLS in following aspects. First, because MPLS labels only have local significance, it requires a dynamic Label Distribution Protocol (LDP) for label assignments. In contrast, XPath path IDs are unique, and we do not need such a signaling protocol. Second, MPLS is based on exact matching (EM) and thus MPLS labels cannot be aggregated, whereas XPath is based on longest prefix matching (LPM) and enables more efficient routing table compression. Furthermore, MPLS is typically used only for traffic engineering in core networks instead of application-level or flow-level path control. In addition, it is reported [6, 22] that the number of tunnels that existing MPLS routers can support is limited.

SPAIN [30] builds a loop-free tree per VLAN and utilizes multiple paths across VLANs between two nodes, which increases the bisection bandwidth over the traditional Ethernet STP. However, SPAIN does not scale well because each host requires an Ethernet table entry per VLAN. Further, its network scale and path diversity are also restricted by the number of VLANs supported by Ethernet switches, e.g., 4096.

PAST [40] implements a per-address spanning tree routing for data center networks using the MAC table. PAST supports more spanning trees than SPAIN, but PAST does not support multi-paths between two servers, because a destination has only one tree. This is decided by the MAC table size and its exact matching on flat MAC addresses.

Both SPAIN and PAST are L2 technologies. Relative to them, XPath builds on L3 and harnesses the fast-growing IP LPM table of commodity switches. One reason we choose IP instead of MAC is that it allows prefix aggregation. It is worth noting that our XPath framework contains both SPAIN and PAST. XPath can express SPAIN's VLAN or PAST's spanning tree using CPF, and it can also arrange paths using DPF and perform path ID encoding and prefix aggregation for scalability.

Finally, there are various DCN routing schemes that come with specific topologies, such as those introduced in Fattree [4], PortLand [31], BCube [18], VL2 [17], ALIAS [43], and so on. For example, PortLand [31] leverages Fattree topology to assign hierarchical Pseudo-MACs to end hosts, while VL2 [17] exploits folded Clos network to allocate location-specific IPs to ToRs. These topology-aware addressing schemes generally benefit prefix aggregation and can lead to very small routing tables, however they do not enable explicit path control and still rely on ECMP [31] or Valiant Load Balancing (VLB) [17] for traffic spreading over multiple paths. Relative to them, XPath enables explicit path control for general DCN topologies.

7 Conclusion

XPath is motivated by the need for explicit path control in DCN applications. At its very core, XPath uses a path ID to identify an end-to-end path, and pre-installs all the desired path IDs between any s-d pairs into IP LPM tables of commodity switches using a two-step compression algorithm. Through extensive evaluation and implementation, we show that XPath is scalable and easy to implement with existing commodity switches. Finally, we used testbed experiments to show that XPath can directly benefit many popular DCN applications.

Acknowledgements

This work was supported by the Hong Kong RGC ECS 26200014 and China 973 Program under Grant No. 2014CB340303. Chang Lan and Hongze Zhao were interns with Microsoft Research Asia when they worked on this project. We would like to thank our shepherd George Porter and the anonymous NSDI reviewers for their feedback and suggestions.

References

[1] Arista 7050QX. http://www.aristanetworks.com/media/system/pdf/Datasheets/7050QX-32_Datasheet.pdf.

[2] H. Abu-Libdeh, P. Costa, A. Rowstron, G. O'Shea, and A. Donnelly, "Symbiotic Routing in Future Data Centers," in *SIGCOMM*, 2010.

[3] J. H. Ahn, N. Binkert, A. Davis, M. McLaren, and R. S. Schreiber, "HyperX: Topology, Routing, and Packaging of Efficient Large-Scale Networks," in *SC*, 2009.

[4] M. Al-Fares, A. Loukissas, and A. Vahdat, "A Scalable, Commodity Data Center Network Architecture," in *ACM SIGCOMM*, 2008.

[5] M. Al-Fares, S. Radhakrishnan, B. Raghavan, N. Huang, and A. Vahdat, "Hedera: Dynamic Flow Scheduling for Data Center Networks," in *NSDI*, 2010.

[6] D. Applegate and M. Thorup, "Load optimal MPLS routing with N + M labels," in *INFOCOM*, 2003.

[7] H. Ballani, P. Costa, T. Karagiannis, and A. Rowstron, "Towards Predictable Datacenter Networks," in *SIGCOMM*, 2011.

[8] T. Benson, A. Anand, A. Akella, and M. Zhang, "MicroTE: Fine Grained Traffic Engineering for Data Centers," in *CoNEXT*, 2010.

[9] I/O Characteristics. http://docs.aws.amazon.com/AWSEC2/latest/UserGuide/ebs-io-characteristics.html.

[10] K. Chen, C. Guo, H. Wu, J. Yuan, Z. Feng, Y. Chen, S. Lu, and W. Wu, "Generic and Automatic Address Configuration for Data Centers," in *SIGCOMM*, 2010.

[11] K. Chen, A. Singla, A. Singh, K. Ramachandran, L. Xu, Y. Zhang, X. Wen, and Y. Chen, "OSA: An Optical Switching Architecture for Data Center Networks with Unprecedented Flexibility," in *NSDI*, 2012.

[12] M. Chowdhury, M. Zaharia, J. Ma, M. Jordan, and I. Stoica, "Managing Data Transfers in Computer Clusters with Orchestra," in *SIGCOMM'11*.

[13] Cisco, "Data center: Load balancing data center services," 2004.

[14] P. T. Darga, K. A. Sakallah, and I. L. Markov, "Faster Symmetry Discovery using Sparsity of Symmetries," in *45st DAC*, 2008.

[15] Bidirectional Forwarding Detection. http://www.cisco.com/c/en/us/td/docs/ios/12_0s/feature/guide/fs_bfd.html.

[16] R. Draves, C. King, S. Venkatachary, and B. Zill, "Constructing optimal IP routing tables," in *INFOCOM*, 1999.

[17] A. Greenberg, J. R. Hamilton, 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 *ACM SIGCOMM*, 2009.

[18] 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 *SIGCOMM*, 2009.

[19] 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 *CoNEXT*, 2010.

[20] C. Guo, H. Wu, K. Tan, L. Shi, Y. Zhang, and S. Lu, "DCell: A scalable and fault-tolerant network structure for data centers," in *SIGCOMM'08*.

[21] B. Heller, S. Seetharaman, P. Mahadevan, Y. Yiakoumis, P. Sharma, S. Banerjee, and N. McKeown, "ElasticTree: Saving Energy in Data Center Networks," in *NSDI*, 2010.

[22] C.-Y. Hong, S. Kandula, R. Mahajan, M. Zhang, V. Gill, M. Nanduri, and R. Wattenhofer, "Achieving high utilization using software-driven WAN," in *ACM SIGCOMM*, 2013.

[23] C. Hopps, "Analysis of an Equal-Cost Multi-Path Algorithm," *RFC 2992*, 2000.

[24] Broadcom Strata XGS Trident II. http://www.broadcom.com.

[25] Provisioned I/O-EBS. https://aws.amazon.com/ebs/details.

[26] H. Liu, X. Wu, M. Zhang, L. Yuan, R. Wattenhofer, and D. Maltz, "zUpdate: Updating Data Center Networks With Zero Loss," in *ACM SIGCOMM*, 2013.

[27] V. Liu, D. Halperin, A. Krishnamurthy, and T. Anderson, "F10: A Fault-Tolerant Engineered Network," in *NSDI*, 2013.

[28] B. D. McKay, "Practical graph isomorphism," in *Congressus Numerantium*, 1981.

[29] N. McKeown, T. Anderson, H. Balakrishnan, G. Parulkar, L. Peterson, J. Rexford, S. Shenker, and J. Turner, "OpenFlow: Enabling innovation in campus networks," *ACM CCR*, 2008.

[30] J. Mudigonda, P. Yalagandula, and J. Mogul, "SPAIN: COTS Data-Center Ethernet for Multipathing over Arbitrary Topologies," in *NSDI*, 2010.

[31] R. N. Mysore, A. Pamboris, N. Farrington, N. Huang, P. Miri, S. Radhakrishnan, V. Subramanya, and A. Vahdat, "PortLand: A Scalable Fault-Tolerant Layer 2 Data Center Network Fabric," in *SIGCOMM*, 2009.

[32] NVGRE. http://en.wikipedia.org/wiki/NVGRE.

[33] Y. Peng, K. Chen, G. Wang, W. Bai, Z. Ma, and L. Gu, "HadoopWatch: A First Step Towards Comprehensive Traffic Forecasting in Cloud Computing," in *INFOCOM*, 2014.

[34] Announcing provisioned IOPS. http://aws.amazon.com/about-aws/whats-new/2012/07/31/announcing-provisioned-iops-for-amazon-ebs/.

[35] E. Rosen, A. Viswanathan, and R. Callon, "Multiprotocol Label Switching Architecture," *RFC 3031*, 2001.

[36] Source Routing. http://en.wikipedia.org/wiki/Source_routing.

[37] Boolean satisfiability problem. http://en.wikipedia.org/wiki/Boolean_satisfiability_problem.

[38] J.-Y. Shin, B. Wong, and E. G. Sirer, "Small-World Datacenters," in *ACM SoCC*, 2011.

[39] A. Singla, C.-Y. Hong, L. Popa, and P. B. Godfrey, "Jellyfish: Networking Data Centers Randomly," in *NSDI*, 2012.

[40] B. Stephens, A. Cox, W. Felter, C. Dixon, and J. Carter, "PAST: Scalable Ethernet for Data Centers," in *CoNEXT*, 2012.

[41] Graph vertex coloring. http://en.wikipedia.org/wiki/Graph_coloring.

[42] VXLAN. http://en.wikipedia.org/wiki/Virtual_Extensible_LAN.

[43] M. Walraed-Sullivan, R. N. Mysore, M. Tewari, Y. Zhang, K. Marzullo, and A. Vahdat, "ALIAS: Scalable, Decentralized Label Assignment for Data Centers," in *SoCC*, 2011.

[44] X. Wu, D. Turner, G. Chen, D. Maltz, X. Yang, L. Yuan, and M. Zhang, "NetPilot: Automating Datacenter Network Failure Mitigation," in *SIGCOMM*, 2012.

[45] D. Xie, N. Ding, Y. C. Hu, and R. Kompella, "The Only Constant is Change: Incorporating Time-Varying Network Reservations in Data Centers," in *SIGCOMM*, 2012.

Increasing Datacenter Network Utilisation with GRIN

Alexandru Agache, Razvan Deaconescu and Costin Raiciu
University Politehnica of Bucharest

Abstract

Various full bisection designs have been proposed for datacenter networks. They are provisioned for the worst case in which every server sends flat out and there is no congestion anywhere in the network. However, these topologies are prone to considerable underutilisation in the average case encountered in practice. To utilise spare capacity we propose GRIN, a simple, cheap and easily deployable solution that simply wires up any free ports datacenter servers may have. GRIN allows each server to use up to a maximum amount of bandwidth dependent on the number of available ports and the distribution of idle uplinks in the network. Our evaluation found significant benefits for bandwidth-hungry applications running over our testbed, as well as on 1000 EC2 instances. GRIN can be used to augment any existing datacenter network, with a small initial effort and no additional maintenance costs.

1 Introduction

Datacenter networks are provisioned for peak load, as operators want performance guarantees even when the network is highly utilised. At the extreme, operators provision their network to fully support any possible traffic pattern: the network core is guaranteed never to become a bottleneck, regardless of the traffic patterns generated by the servers; such networks are said to provide *full-bisection bandwidth.* FatTree [3] and VL2 [9] are full-bisection datacenter topologies deployed in production networks. A high profile example is Amazon's EC2 cloud that was using a topology resembling VL2 for their regular instances until recently (see Section 2 in [18]), and are now deploying 10Gbps FatTrees.[1]

Measurement studies show that datacenter networks are underutilised: the traffic has on-off patterns leading many links to run hot for certain periods of time, while even more links are idle, leaving the network core underutilised most of the time [9, 13, 6]. Datacenters heavily rely on the concept of resource pooling: different applications' workloads are multiplexed onto the hardware, and any application can in principle expand to utilise as many resources as it needs as long as there is any available capacity left. Resource pooling does not apply to datacenter networks: although the core is underutilised, hosts cannot take advantage because they are often bottlenecked by their NICs.

In this paper we set out to make datacenter networks better at resource pooling, which would increase their average utilisation and provide better performance per dollar. The obvious solution is to use bigger links between servers and Top-of-Rack switches: in a 10Gbps full-bisection network this means either 40Gbps links or using multiple 10Gbps links (i.e. multihoming). This technique improves performance: when few servers are sending data, they can send many times faster. Unfortunately, it is also very expensive: 40Gbps links are not commodity yet and switches supporting them have small port densities. Multihoming seems more viable economically, but even dual-homing requires doubling the number of ToR switches in the network.

We propose GRIN, a simple change to existing datacenter networks where any free port existing in any *server* is connected to a free port of another server in the same or a neighboring rack. Servers can then communicate using a path provided by the original topology, or via one of the servers they are directly connected to. GRIN can function seamlessly over existing datacenter networks, and the best case benefits are obvious: for every additional port used, a server can send 1 or 10 Gbps more traffic if the neighbour's uplink is idle. As modern NICs are often dual or quad-ports, these additional interfaces should be already available in most servers on the market. [2]

We tested our GRIN implementation with various workloads and applications, using both a small local cluster and on 1000 EC2 hosts. GRIN can significantly increase end-to-end application performance; a 2-port GRIN setup speeds up HDFS, NFS, Spark and Cassandra by a factor of two or more in certain scenarios. While unmodified applications can benefit from GRIN, these gains depend on the traffic pattern of the neighbouring servers. Many datacenter apps run on multiple nodes, and scheduling work across these is already an important component for performance. We have modified a few such apps to be GRIN-aware when taking their scheduling decisions (namely HDFS, Hadoop and Spark [25]). By jointly optimising addressing, routing and applications, GRIN can achieve performance similar to multihoming.

2 Problem Statement

To understand why full-bisection networks are underutilised most of the time, we measure the network utili-

[1] Private conversation with Amazon engineers.

[2] Our brief survey shows most adapters on the market are dual and quad port, and price per port decreases significantly for multi-port NICs.

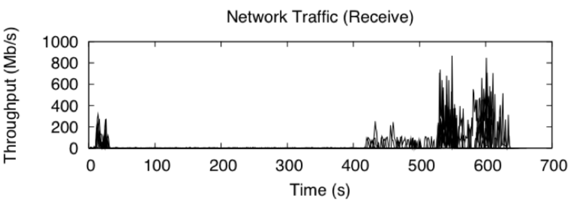


Figure 1: Network utilisation of a simple Map/Reduce job

sation of a small cluster of ten servers connected to non-blocking switch and running a map-reduce job, a typical datacenter application [8]. The servers run Hadoop word-count over a collection of web pages (50GB) and they store at replication level 3. We plot the network throughput measured for one server in Figure 1.

In the map phase, there will be a small percentage of tasks whose data is non-local ([26] estimates 1%-10%), thus requiring filesystem reads from other servers, which will be bottlenecked by the host NIC capacities, assuming appropriate storage provisioning. The shuffle phase will move the data generated by the mappers to the reducers. The shuffle phase is notoriously bandwidth hungry, but this depends on the number of reducers. In the worst case, all servers are reducers and download data at the same time, leading to an all-to-all traffic pattern that fully utilises the network core—in fact, this is the main motivation given for building full-bisection networks [3]. In practice, the number of reducers is an order of magnitude smaller then the number of mappers, and the shuffle phase starts earlier for some servers, thus the core network utilisation will be a lot smaller; still, some reducers will be bottlenecked by their servers' NIC. Finally, the output of the reduce phase is written to disk leading to point-to-point transfers, again bottlenecked by the host NIC.

We want to change existing topologies to allow hosts to utilise as much of the idle parts of the network as possible when other hosts are not active (in the map or output phase of the job above, for example). Good solutions share the following properties:

- **Ability to scale:** cost is a major factor that determines what is feasible to deploy in practice. Using more server ports should increase performance and incur little to no additional costs.
- **Fairness and isolation:** access to neighbour links must be mediated such that each server gets a fair share of the total bandwidth. Misbehaving servers should be penalized, and they should not adversely affect the performance of the network.
- **Widely applicable:** it should be possible to apply the solution to existing or future networks.
- **Incrementally deployable:** it should be possible to deploy the solutions on live datacenter networks with minimal disruption. This implies hardware or software changes to the network core (including routing algorithms) are out-of-scope. Further, upgrading only a subnet should bring appropriate benefits.

Barring extensive changes to the original topology, the most straightforward solution is to multihome servers by using additional TOR switches. We add a TOR switch for every additional server port (see Fig.2b), so that each server is connected to each of the multiple TOR switches from its rack. In order to keep the rest of the topology unchanged, we evenly divide the uplinks of the original TOR switch between all the local TOR switches. The resulting topology is oversubscribed, but now each server can potentially use much more bandwidth. Multihoming brings additional costs in terms of switching equipment, rack-space, energy usage and maintenance. As every additional server port could require an extra switch, this solution does not scale well with the number of server ports.

3 GRIN

Our solution is to interconnect servers directly using additional ports (some of which are already installed, and effectively free), while keeping the original topology unchanged. Each pair of servers that are directly connected in this manner become neighbours. Intuitively, when a server does not need to use its main network interface, it may allow one or more of its neighbours to "borrow" it, by forwarding packets received from them (or packets addressed to them) to their final destination. This solution is depicted in Figure 2c and we call it GRIN.

When a server wishes to transmit, it can use both its uplink and the links leading to its neighbours. Conversely, the destination can be reached through both its uplink and via its neighbours. We call the links used to interconnect servers, horizontal (or GRIN) links, and reserve the term uplinks for those that connect servers to the switch in the original topology. The network interface where the uplink is connected becomes the primary interface of the server, while the others are considered to be secondary interfaces. If every server has n GRIN neighbours, we say that the *degree* of the GRIN topology is equal to n.

Cabling. The baseline GRIN implementation relies on servers being interconnected within the same rack. This is the cheapest solution, and should work at even the highest link speeds. From a performance point of view, it is best to connect those servers that usually do not need to access the network at the same time, otherwise interconnection will not bring major gains beyond improving local throughput. Traditionally, distributed applications tend to localize traffic within racks or pods (multiple racks) because inter-pod bandwidth used to be scarce. In such cases

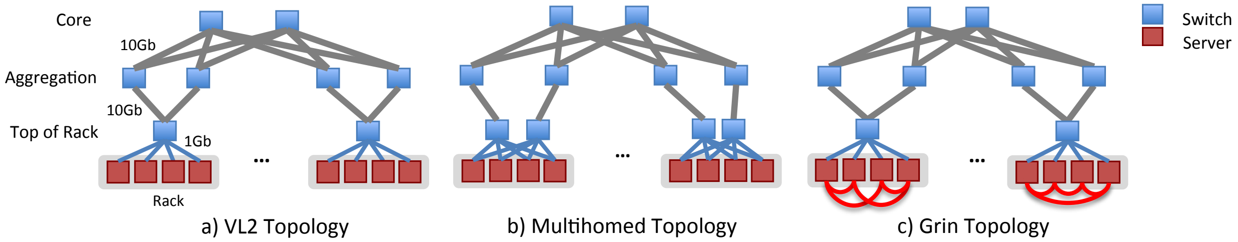


Figure 2: Enhancing a VL2 topology to improve network utilisation

and when nearby servers run the same distributed application, we can enhance the application's scheduler to become GRIN aware (see §3.4). Whenever possible, we can also connect servers from neighbouring racks. Cross-rack cabling is more complex to wire but ensures better fault tolerance to ToR switch failures.

Path Length. Most datacenter topologies have multiple equal cost paths between servers. With GRIN, the set of paths can grow to include elements consisting of increasingly large number of hops. When choosing a path between two random servers, we can select (1) one of the paths available in the original topology, (2) a path that consists entirely of GRIN links, and (3) any number of intermediate servers, and form a path using the concatenation of all intermediate paths.

A compromise must be found between limiting path length and trying to make the most of the available network capacity. One limitation is the fact that any path should traverse at most two uplinks: once in the full-bisection bandwidth network core, there is no reason for traffic to be bounced via another "waypoint" server. Thus, in a GRIN topology a path consists of three segments:

- the first horizontal segment, which is a group of horizontal links going from the source to the server whose uplink is used to reach the first switch
- the path taken from the first to the second uplink through the original network core
- the second horizontal segment, which also consists of a number of GRIN links and goes from the second uplink to the destination server

A horizontal segment can also be empty; for example, any path that was also available in the original topology does not include any horizontal links. We use the term $horizontal_n$ routing (or h_n routing) to describe the fact that in a given GRIN topology we are only interested in paths which have horizontal segments of length at most n; by this definition, the original topology uses h_0 routing.

3.1 A strawman design

For GRIN to work, it needs five key components:

- An algorithm to assign addresses to horizontal GRIN interfaces. We term these *secondary addresses*.
- Techniques to enable servers to discover secondary addresses of other servers, so that they can use them to send traffic.
- Ability to efficiently route traffic to secondary addresses via the appropriate neighbour.
- Ability to create paths spanning multiple horizontal GRIN links.
- A mechanism to spread traffic over the multiple paths

Designing all these seems simple at first sight, and a strawman solution is the following. First, assign datacenter-unique addresses to secondary interfaces in a different subnet from the uplinks to allow servers to distinguish primary and secondary addresses. To enable discovery of secondary addresses, simply use DNS, assuming it is already deployed in the datacenter: register a new "A" record for every secondary interface of server s in the DNS entry for s. Servers discover secondary interfaces by running a DNS lookup.

Routing traffic to a secondary address via a neighbour is trickier. The routing system needs to be informed of the neighbours reachable via a server, and the straightforward solution is to run the datacenter routing protocol (e.g. OSPF) all the way down to the servers, and have neighbour addresses announced in the routing system. This solution is unwieldy: link-state protocols such as OSPF do not scale well beyond a few thousand routers, and pushing servers into OSPF breaks this boundary. Secondly, we need to significantly reconfigure the datacenter routing protocol. Even assuming the routing system can be upgraded, we still have the problem of routing via multiple horizontal links. For this to work, the source must be able to specify a list of intermediary addresses for the packet, and the simplest solution is to use loose source routing. LSR support exists in most operating systems, but it raises performance problems: it does not work well with hardware offloading on some NICs, and LSR packets could even find themselves on the slow path of routers.

One last mechanism is needed to spread traffic over the available paths. As TCP is the de-facto standard transport protocol in datacenter, the straightforward solution is to pin each connection to one path (i.e. round robin), but there is a danger than a neighbour link is highly congested. In such cases we should move traffic back to the uplink quickly, which is easier said than done—to ensure proper routing we need to change the addresses in the packets which will break the TCP connection. To solve this problem we need to either change the host stack (add some sort of mobility support) or modify the applications.

Towards a solution. The main conclusion arising from the strawman solution above is that dealing with the subproblems in isolation is inefficient because one solution affects many others. For instance, independent addressing forces us to use unscalable routing schemes, and neighbour discovery mechanisms might be redundant if we have a mobility solution in the host stack.

That is why we take a holistic approach to designing these mechanisms, co-optimising them to enable a cheaper and easier to deploy solution. Our first insight is that we could avoid source routing if we limit ourselves to one horizontal hop after the source and one before the destination (h_1 routing). In a h_1 network, the sender can steer outgoing traffic over either the uplink or GRIN links by simply placing them on the appropriate interface. To route via a neighbour of the destination we leverage our addressing scheme (described below). On the other hand, h_1 routing appears to have the least potential of actually using spare network capacity. How much of an improvement, if any, can be achieved by increasing the length of horizontal segments? The results of our evaluation in section 5.1, show that h_1 routing utilises most of the capacity, while being the cheapest from a forwarding point-of-view. That is why we focus on h_1 routing alone in our solution.

The two main building blocks of GRIN, described next, are an addressing scheme that works without changes to the routing system, and using Multipath TCP to efficiently utilise network capacity.

3.2 GRIN addressing

To solve the routing system scalability issues, we relate the assignment of secondary addresses with the addresses of the uplinks, as follows. All network addresses $addr$ are split into two meaningful groups of bits: the most significant 24 bits are the server identifier, $I(addr)$, which must be unique in the datacenter. The last 8 bits represent the GRIN identifier $G(addr)$, which is set to 1 for primary interfaces and larger values for secondary interfaces.

The address of a secondary interface has the same server identifier as the address of the primary interface of the neighbour it connects to; the only difference is the GRIN identifier. Formally, for any server s, let up_s be the address used by its primary interface and n_s be the number of neighbours s has connected with already; n_s is zero when we start the address assignment algorithm. To connect server t to server s, the address of the secondary interface is $grin_{t,s}$ and has the following structure:

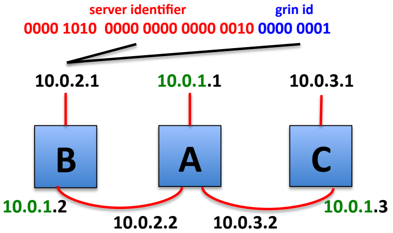


Figure 3: Grin Address Assignment Algorithm

$$I(grin_{t,s}) = I(up_s) \qquad G(grin_{t,s}) = 2 + n_s$$

Given a secondary address, we can infer the primary address of its neighbour by merely substituting the least significant byte with 1. This scheme also provides an easy way to differentiate between multiple secondary interfaces connected to same neighbour s, because n_s will increase after each subsequent interconnection.

Consider the example in Figure 3, where there are two additional network ports available on each server. Initially we choose to connect servers A and B. Since up_A = 10.0.1.1 and up_B = 10.0.2.1, we will assign the address 10.0.2.2 to $grin_{A,B}$ and 10.0.1.2 to $grin_{B,A}$ (because both n_A and n_B are 0 initially). Now $n_A = 1$, and $n_C = 0$, so if we interconnect servers A and C $grin_{A,C}$ will be 10.0.3.2, and $grin_{C,A}$ will be 10.0.1.3.

Routing is greatly simplified with this addressing scheme: a simple router configuration change should be enough for most networks to be adapted to GRIN. In fact, many networks already assign subnets instead of individual addresses to servers, to support direct addressing for virtual machines running on them. Finally, the servers will need to be configured to forward traffic they receive for their neighbours. Our addressing scheme sacrifices 8 bits for the GRIN identifier, supporting more than 250 horizontal links. The remaining 24 bits can uniquely identify up to 16 million servers. Fewer bits can be used for GRIN links (e.g. 4) to support a larger number of servers.

3.3 Packet Forwarding

To implement GRIN we can reuse forwarding support provided by modern OSes. Linux, for instance, peaks at a rate of about 570Kpps in our tests. This is good enough for gigabit links, or even at 10 Gbps with jumbo frames, but cannot really keep up as NICs become faster.

Can we do better? When processing packets for its GRIN neighbours, a server is fulfilling three main

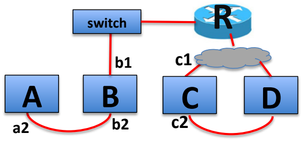


Figure 4: Simple GRIN1 Setup

functions: identification of packets intended for another server, rewriting some header fields and forwarding the result. For packet identification, we can leverage hardware filtering capabilities present in modern NICs which allow packets to be received on different queues based on various discriminants, such as destination address. With IP forwarding, each server must write at least the MAC source and destination addresses in the packet. As no hardware support exists for IP forwarding in commodity NICs, this operation must be performed in software.

In theory, GRIN allows us to avoid this extra work by using *bridging* instead of forwarding. In this context, bridging means simply passing a packet from one interface to another, without doing any kind of software processing on it. The origin server knows the MAC address of the next IP hop for a packet. In an L2 network this is the primary interface of the destination (if the packet is heading for a primary address) or the primary interface of one of the destination's neighbours. For a L3 network, the next hop is the designated first router. In both cases, we use ARP to find the proper MAC address, as the IP address is already known (directly from the packet for L2 and by configuration for L3).

This concept is presented in Figure 4: servers A and B are neighbours, and the default gateway for B's uplink traffic is router R. When A sends a packet to C, with basic forwarding the packet will have the destination IP address $D_{ip}(p) = c_1$ and destination MAC address $D_{mac}(p) = mac(b_2)$. The latter will change over the following hops, first to $mac(r_1)$ and, eventually, $mac(c_1)$. With bridging, the packet leaves A with $D_{ip}(p) = c_1$ and $D_{mac}(p) = mac(r_1)$; it can find $mac(r_1)$ by sending an ARP request. B can simply pass the packet to the uplink upon reception; the original contents are enough to steer it toward C, since the router knows how to reach c_1.

If p is sent to a secondary interface of C, for example c_2, that is connected to a server D, then there is one more significant step to be mentioned. The packet will leave A, and reach R as in the previous case. The routers are configured to send packets with secondary destination addresses to the appropriate neighbour (D for c_2, in this example). Once p reaches server D, it can be placed directly on the link which is connected to c_2, based on the destination IP address. As C receives p, it will have to ignore the incorrect destination MAC address (the last change to this field was made by the router before D). We consider this behaviour to be acceptable, as secondary interfaces only receive packets from their neighbours.

In summary, h_1 routing allows GRIN to not only avoid source routing, but also IP forwarding: intermediate servers can just copy packets between interfaces without "touching" them. Modern commodity 10Gbps NICs (e.g. based on the popular Intel 82599 chipset) already have a simple hardware switch, but it can only move frames between different queues of a single NIC in the *tx*→*rx* scenario; GRIN needs to pass packets from one *rx* queue of an interface to a *tx* queue of another interface (the two interfaces will likely belong to the same multi-port card). With this in place, we could eliminate forwarding overhead altogether. Until hardware support is available, we continue to rely on Linux forwarding for our implementation, described in Section 4, where we also present a prototype that shows the viability of the bridging solution.

3.4 Efficiently using GRIN Topologies

TCP binds a connection to a single NIC, so several parallel TCP connections are needed to utilise the GRIN links available via neighbours. However, according to measurements there are few large ("elephant") flows at any time on a given server [9]; simply adding more NICs should not bring significant performance benefits for most applications over regular TCP. Changing every application to spread data over multiple TCP connections is not feasible: such changes are complex, and there is too much to be changed.

GRIN enables unmodified apps to opportunistically increase their network performance by using Multipath TCP [19]. MPTCP allows a transport connection to grow beyond one NIC (the uplink) and effectively utilise the neighbours' spare capacity. MPTCP also provides dynamic load-balancing across paths: when a path is congested (e.g. via a neighbour) traffic will be automatically shifted to less congested paths [24]. While MPTCP is just one of many possible multi-path forwarding designs, we consider it an ideal enabler for a lightweight implementation, that does not warrant any changes to user applications or the network itself (beyond horizontal links).

Path selection algorithm. Let's see how MPTCP works over a GRIN network. In the example from Fig. 3, assume the only GRIN links are those shown, and that server B wants to send data to server C. The connection begins like regular TCP between the primary interfaces of the two servers. After the initial handshake is complete, server B will be notified via the MPTCP address advertisement mechanism of any additional addresses it can use to contact server C—this mechanism enables neighbour discovery without needing DNS at all.

The set of additional addresses for server C will consist of only one element, $10.0.1.3$. Server B will then attempt to establish a full mesh of subflows between its own addresses (both primary and secondary) and those just received. Thus, we can rely on MPTCP to deal with path selection, and only add small tweaks to the process, as mentioned in Section 4. How soon should we use the additional paths? The easiest answer is to setup additional subflows as soon as address advertisment completes, but this might not always be desirable, especially for short flows—sending few packets over a different subflow raises the probability of a timeout, in case one of the packets gets lost. To protect small flows our implementation uses a configurable threshold, sending all bytes below that via the uplink (100KB for a 1Gbps network); MPTCP will use the other subflows thereafter.

3.5 GRIN-aware applications

A downside of opportunistic usage is the probability than two neighbours will be using their uplinks at the same time; the busier the network is, the higher this probability. However, most datacenter applications have centralized schedulers that decide how to partition the work across the many workers in the system. We can gain performance comparable to multihoming solutions without the associated costs if we modify application schedulers to take into account GRIN links. We have implemented such optimisations for Hadoop, HDFS and Spark [25].

Scatter-gather applications (such as web search) open multiple TCP connections from a frontend server to multiple backend servers. They are bottlenecked by the frontend server's NIC, and susceptible to the "incast" problem [22]. Scatter-gather apps can be easily optimised for GRIN: they just should disable MPTCP and "pin" different TCP flows onto the different available paths for best performance. We have optimised a synthetic scatter-gather frontend server to spread its connections evenly across the GRIN neighbours, thus increasing the total buffer size available to the frontend. We present experimental results for GRIN-aware applications in § 5.6.

4 Implementation

The GRIN implementation works with a MPTCP-enabled Linux kernel and mainly deals with address assignment to secondary interfaces in user-space. We have also made minor changes to the MPTCP kernel in order to improve performance and prevent some unwanted interactions.

The GRIN addressing scheme allows us to use any routing mechanism that was already in place in the original topology, as long as the original addresses can be adapted to the new structure. The assignment itself relies on pre-existing mechanisms, e.g. DHCP.

To automatically configure secondary interfaces, we have implemented a simple server that runs on every computer. It only serves requests that arrive on GRIN interfaces, and its primary functionality is the dissemination of proper secondary addresses to neighbours. After the endpoints of a horizontal link exchange addresses in this manner, each server also adds the required information to the local routing tables. There are two such entries needed per neighbour: one to designate it as the default gateway for all traffic leaving that particular secondary interface, and another to state that the address of the remote endpoint is reachable via the same interface. Additionally, we use Proxy ARP to make servers reply to ARP requests for their neighbours' secondary interfaces, while also making them ignore queries for their own such interfaces.

The first change we made to the MPTCP kernel is related to subflow initiation. Establishing a full mesh, especially for higher GRIN degrees, would setup a very large number of subflows, which is often undesirable. The default behavior for GRIN is to establish the smallest number of subflows such that every horizontal link is used at least once. Thus, in a GRIN topology with degree n, MPTCP will establish n additional subflows. There is also the option of specifying a certain number of subflows, which are going to be selected at random in a manner consistent with the goals of the default behaviour. Another modification was to adjust the MPTCP subflow selection process, which decides what subflow to use when sending each particular packet. In most situations, we want to send data using the direct subflow whenever its congestion window allows it. MPTCP selects the subflow with the smallest RTT when multiple subflows could send a packet, but the RTT estimation alone is noisy and might sent packets on horizontal links even when the uplink is idle. That is why we added bias in favor of the direct connection: the estimated RTT of the uplink is halved for comparison purposes.

Bridged implementation. We have also implemented the prototype of a bridged GRIN1 topology that uses netmap [20] as a stand-in for the missing hardware functionality. Outgoing packets that are heading to a primary interface receive no special treatment. For all others, we make sure that the MAC destination address field contains the L2 address of the proper gateway. For both primary and secondary NICs, we use ethtool to enable ntuple filtering and add filters that make sure any packet destined for the local server arrives on *rx* queue 0, while all others are received on queue 1. Finally, the netmap bridge ensures that packets received on queue 0 of each NIC are sent to the host TCP stack and packets coming from the host stack are sent using *tx* queue 0. Also, packets received on *rx* queue 1 of any interface are simply sent to *tx* queue 1 of the other interface. This could easily be extended to work with higher GRIN degrees by using an additional *rx/tx* pair of queues for each secondary interface added.

For the setup in Figure 4, when a packet p going from a_2 to c_1 reaches B, it will be placed in *rx* queue 1, because $D_{ip}(p) \neq ip(b_2)$, the netmap bridge will transfer it to *tx* queue 1 of b_1. The switch will direct the packet towards r_1, based on the destination MAC address, and from R it will make its way to the destination. We used the routing implementation in our evaluation because netmap does not support hardware offload when exchanging packets with the Linux TCP stack, affecting performance.

5 Evaluation

This section starts with an analysis of the effect h_n routing can have on GRIN's ability to utilise spare capacity—the result led us to choose h_1 routing and build our solution around it. We continue by evaluating the performance benefits of GRIN, both in synthetic scenarios and for real applications. We also include an assessment of the potential negative impact that GRIN may have in terms of fairness, latency and forwarding overhead. This can be especially important for opportunistic usage.

5.1 How many horizontal hops are needed?

The advantages of h_1 routing in terms of reduced complexity are obvious, but is there anything we lose by using it? We intend to find out if there is any correlation between the maximum allowed path length and the amount of capacity that can be discovered. We model a GRIN topology as racks of computers connected to a single, sufficiently large switch. The computers are interconnected using a variable number of GRIN links (1 to 6), and the entire setup is represented as a directed graph.

We are interested in the maximum network capacity that can be utilised in each case. Given a set of source-destination pairs , we use GLPK [1] to solve the maximum multi-commodity flow problem, which gives the optimal solution and is a hard upper bound on usable capacity. We also solve a couple of specializations to MCF in which we restrict the number of horizontal hops to emulate the best we can do with the corresponding h_n strategy. Due to computational complexity, our network model consisted of six twenty server racks. We run experiments varying the GRIN degree, the maximum number of horizontal hops (h_1, h_2 and no limit), and the traffic pattern:

- *permutation traffic*: each active server sends data to a single destination, and each destination receives data from a single source.
- *group traffic*: servers are randomly assigned to groups such that every group contains the same number of servers. One server is randomly chosen from each group as the destination for the other group members.
- *all-to-all traffic*: each active server sends data to every other active server.
- *random traffic*: the endpoints of every connection are chosen at random.

The results show the total flow is proportional to both the number of active connections and the GRIN degree. There are two interesting observations. First, the difference between h_1 and the optimal solution is at most 32% for GRIN6 and permutation traffic. On average, across all experiments, the difference between h_1 and optimal is just 7%. Also, the difference between the optimal solution and h_2 is seldom more than 1%.

Unfortunately, optimal placement of flows to paths is impossible to achieve, so it's quite hard to form an expectation of real-world behavior based on these results. Any MPTCP connection will only use a limited number subflows in order to prevent performance degradation [24]. Also, it's not usually possible to make informed decisions about flow placement in real time; instead flows are spread over multiple random paths and congestion control balances traffic to get the most out of the network.

To capture these effects, we devised another performance estimation procedure: for a given network and traffic matrix, we start by building the complete set of paths between any source and destination. The length of a path is defined as the number of horizontal segments it contains, with one exception: if a path is made of a single horizontal segment, then its length is zero. The shortest path that goes through the switch is called the direct path. For each connection, we randomly select up to 16 paths (a relatively large number) without replacement and add them to the chosen path set; the direct path is always included. We try to assign the largest possible flow to each element of this set, in ascending order of length. This is done by finding the path segment with the least amount of available capacity, and then using that value to fill the entire path. We applied this method to the same input data as before. We also considered h_3 routing, as without optimal placement, h_2 may no longer be sufficient.

The results are surprising: h_1 provides better results in

	GRIN degree					
%	1	2	3	4	5	6
10	23/23 21/21	30/30 29/29	39/40 37/33	49/52 39/37	55/65 41/39	61/70 43/42
20	41/41 38/38	56/56 50/49	65/70 61/56	80/90 64/60	92/108 66/63	101/118 68/66
30	57/57 52/52	71/71 65/65	85/92 77/72	103/116 81/77	114/136 83/80	118/143 85/84
40	65/65 63/63	88/88 78/78	106/114 91/85	112/131 93/89	131/153 95/93	136/160 97/96
50	75/75 73/73	100/101 88/89	120/128 100/95	122/139 103/99	135/159 105/103	140/163 107/105
60	87/87 83/83	110/110 97/99	119/134 107/103	129/154 109/107	137/169 112/110	141/174 114/113
70	95/95 92/92	121/122 103/106	122/134 112/110	131/158 114/113	140/170 118/117	149/192 120/120

Figure 5: Hop Analysis Results for Permutation Traffic

most experiments, and the increase is larger as the GRIN degree increases. This behaviour reflects the fact that h_2 and h_3 routing increase contention on horizontal links which leads to more collisions. In the MCF analysis, this effect was offset by the very large number of paths used and exhaustive search used by h_2 and h_3.

Fig. 5 shows detailed evaluation results for permutation traffic. The leftmost column represents the percentage of active servers from each experiment. The first row of each cell shows the maximum flow for h_1 and h_2, respectively, as computed by MCF. The second row has values obtained using our alternative evaluation procedure (h_1 / h_2). We focus on permutation traffic as it exhibits the largest differences in both cases (worst relative behaviour for h_1).

The maximum flow is not included because it is very well approximated by h_2. Note that the total flow values can be larger than 120 (the total number of uplinks) because some connections can be established over entirely horizontal paths. The largest differences between h_1 and h_2 on the first row appear for GRIN degrees unlikely to be encountered in practice. On the second, there are only a few instances where h_2 is marginally better. We conclude that h_1 is the best solution given our constraints.

For other traffic patterns, the differences between h_2 and h_1 are smaller than with MCF, and h_1 gives better performance when measured with our alternative method.

5.2 Experimental setup

We deployed our implementation on a small local cluster of ten servers directly connected to a switch to examine real-world application performance. Each server has a Xeon E5645 processor, 16 GB of RAM, a quad-port gigabit NIC (one port is used for management) and a dual-port 10Gbps NIC. In our testbed, we can build 1Gbps GRIN1 and GRIN2 topologies and a 10Gbps GRIN1 topology.

We use both gigabit and ten gigabit networks in our evaluation. Gigabit links to servers are still in wide use today, and GRIN can offer an immediate and much needed increase in performance for deployed networks assuming extra server ports are available. Our ten gigabit tests aim to establish is GRIN is also applicable to newer networks that use 10Gbps links to the servers.

The small size of the testbed prevented us from building a useful multihomed setup; even if the bandwidth constraints could be enforced, we could only have at most two racks of five servers each which would allow communication at double speed with half of the servers in the testbed. An upper bound is provided instead by simply doubling the results obtained in the original setup.

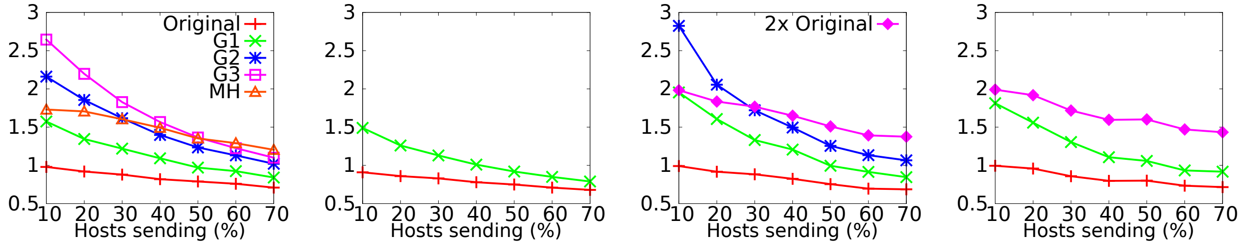
We also deployed a larger GRIN1 topology on 1000 Amazon EC2 c3.large instances that allows us get an impression of our solution's scalability in practice. The instances ran in an Amazon VPC (Virtual Private Cloud), which offers the illusion of an L2 network, with small adjustments to the implementation as proxy ARP did not work in this setting. When dealing with L2 address resolution for secondary interfaces in the kernel, we instead issue an ARP request for the primary address of the corresponding neighbour (which can be easily computed based on the GRIN addressing scheme).

By default, each instance comes with one ENI (elastic network interface), but more (up to three for c3.large) can be added. The first is used for management purposes, the second is the considered the uplink, and the last plays the role of secondary interface. However, unlike regular NICs, all these share the same physical link. We employ dummynet [7] to add bandwidth limitations adding up to less than the maximum available for a single instance, in order to achieve virtual separation. The limit for each interface is set to 100Mbps. At higher speeds, our dummynet configuration led to erratic behaviour.

Finally, to understand the basic properties of GRIN across a wider range of parameters than feasible in practice, we used simulation in *htsim*, a scalable packet level simulator. This has the advantage of giving very precise results and allows us to study reasonably large networks, but doesn't account for factors outside of the transport protocol itself and cannot be used to evaluate applications. Our simulations were based on the same 120 server topology described in the previous section. Increasing network sizes up to tenfold provides qualitatively similar results, however it takes substantially longer to run.

5.3 Basic performance

To understand how GRIN works in practice, we begin our tests with synthetic traffic patterns that we can easily reason about. We use the same patterns described our hop-count evaluation, namely permutation, random, group and all-to-all, and run the experiments in both simulation, on



(a) Simulation, 120 servers (b) EC2, 1000 servers (c) Testbed 1Gbps network (d) Testbed 10Gbps network

Figure 6: GRIN improves performance by 50% to 150% for Random traffic.

Amazon EC2 on 1000 servers and on our local testbed with gigabit and 10 gigabit networks. In all cases we run at least the baseline and GRIN1. We also simulate multihoming, and GRIN3, which we view as an upper bound of the number of ports that GRIN may use in practice.

In Figure 6 we present average per-server throughput results for random traffic as we vary the percent of active servers from 10% to 70%. One thing to keep in mind is that, due to the small size of our local testbed, connections between neighbours will happen more often, and the results will appear better overall. To allow easy comparison between graphs, we normalize the resulting throughput measurements with respect to the best outcome in the original topology (941Mbps for the 1Gbps network, 9960Mbps for 10Gbps and 100Mbps for EC2). Simulation results are normalized by default.

The results show that, as expected, lower percentages of active servers lead to significantly better results for GRIN topologies. Performance improvements are smaller as more servers become active, GRIN2 and GRIN3 performance is better than multihoming until 40% of servers are active, and match it after that. Note that the simulation results are matched very well by EC2 results and are qualitatively similar to the testbed results, giving us confidence in our evaluation. The EC2 results underline the scalability of our solution: a real-life deployment of GRIN1 can run on 1000 servers. Running GRIN at 10Gbps is also worthwhile, doubling the throughput when few servers are active. The results for permutation traffic are similar; the interested reader can refer to [2] for details.

We next turn to all-to-all traffic, a pattern mimicking the shuffle phase of map-reduce. When running this experiment on EC2 we ensure that no server initiates or receives more than 20 concurrent connections to reduce the effects of incast. The results in Figure 7 show that every additional port used with GRIN brings close to 100% performance improvement when few hosts are active. Multihoming is almost always dominated by GRIN2 and GRIN3, however it outperforms GRIN1. As expected, the testbed results are better. The EC2 results accurately track those obtained in simulation.

Finally, the group connection matrix simulates scatter-gather communication. This is the most favorable situation for GRIN topologies, because the large number of sources will fill every link of the receiving server. The results are consistent across both simulation and actual implementation: we get close to the optimal throughput.

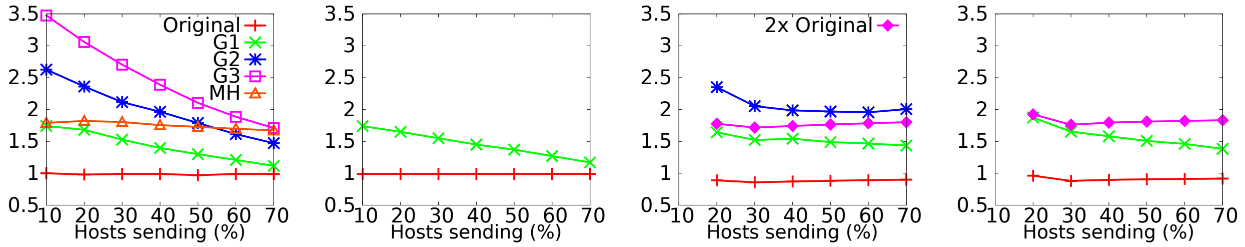
Short flows. We also wanted to find out when GRIN starts to offer benefits if we have fixed-size transfers, assuming there is no contention anywhere in the network. We ran a series of tests using a simple client-server program which requests and then receives a certain number of bytes. The results for the 1Gbps network are shown in Figure 8, and reveal that we need to transmit data on the order hundreds of kilobytes before any sizeable gain becomes apparent for a single connection. That is why we have set the multipath threshold to 100KB in our implementation for this scenario. Also, for smaller transfers (between 15-75K), GRIN may add at most 200μs to the completion time. This issue is caused by the way data is distributed among subflows. At 10Gbps, the threshold increases to 20MB.

5.4 Opportunistic Usage

GRIN can be used opportunistically by simply deploying it and running applications. We deployed a number of real-world applications on a 1Gbps network, where GRIN can have an immediate impact.

The first application is an **NFS** [16] server. Our goal was to measure the time it took to read every file from an exported directory. We varied the file size from one experiment to another, while ensuring their aggregate size was 2GB. As can be seen in Figure 9, throughput improvement is directly proportional to file size. After a certain point, each file is large enough to make the request overhead almost negligible in relation to the actual transfer duration.

Another application we considered is **HDFS**[21]. This is a natural choice for any GRIN setup because it involves handling large amounts of data, so we potentially have a lot to gain in terms of performance. One server was used to host the NameNode, 8 were running DataNodes,



(a) Simulation, 120 servers (b) EC2, 1000 servers (c) Testbed 1Gbps network (d) Testbed 10Gbps network

Figure 7: GRIN improves performance by 80% to 250% in the All-to-all traffic pattern.

and the last one acted as a client. We measured the time needed to transfer a 4GB file from HDFS to local storage. With GRIN1 we got the best possible result, as the file transferred almost twice as fast. The switch to GRIN2 however, did not bring the threefold increase in speed we were hoping for. This was apparently caused by the java process becoming CPU bound. When we requested two transfers in parallel, each process ran on a different core and every network interface was fully utilised.

Next, we wanted to see if GRIN can bring any benefits to **virtual machine migration**. We installed the Xen Hypervisor[5] version 4.2 on several servers running our modified version of the MPTCP kernel. The virtual machine created for the experiments had 4GB of RAM, and its disk image was shared by an network block device server. The metric used during each test was the time required to migrate the VM to another server. Our first attempts, using the xl toolstack, met with failure because it uses ssh to send migration data, which incurs a hefty overhead, so the network is no longer a bottleneck. We got the best results by switching to xm, which uses plain TCP. In this case, using GRIN1 increased migration speed by around 60%, while GRIN2 doubled it.

The last application in our 1Gbps test suite was **Apache Cassandra**[14]. We started a Cassandra cluster consisting of 9 servers, while the last one acted as client. Our goal was to use the Cassandra-stress tool to measure the time required to write a constant amount of data in different circumstances. We used the default values for most parameter, only changing the number of columns to 10, and then varying the size of each column and the numbers of keys inserted such that the total transfer size was around 2GB. The relation between column size and request completion time is presented in Figure 10. Somewhat unsurprisingly, the way parameters combine to determine the amount of data per row is what matters most in terms of performance. The operations complete much more quickly for a smaller number of larger rows. Increasing the number of columns while decreasing the number of rows will also lead to better results, but not to the same extent as using larger rather than more columns.

10Gbps networks. Regular applications don't scale nearly as well as the iperf experiments do at 10Gbps, even with jumbo frames and hardware offload functionality enabled. Thus, we focused on a few apps have high bandwidth requirements and are fast enough to take advantage of it: NFS, HDFS and Spark [25].

With NFS, the server hosts a single 12 GB file which can be transferred by one client in around 10 seconds with GRIN disabled, and a little less than that with one secondary interface enabled. GRIN does not help because the client is CPU-bound. If two clients attempt to transfer the same file *simultaneously* over a GRIN1 network, they both finish in around 10.5 seconds, implying that the server was able to fully utilise both its 10Gbps interfaces.

We used a variable number of Spark workers, connected to a single-node DFS deployment, to count the occurrences of a string in the same 12GB file. At first, GRIN is disabled. A single worker completes the task in 14.5 seconds on average. We need two workers to reduce to time to 10.5 seconds, which is very close to the duration of the data transfer, so we can consider the computation to be finally network-bound. With GRIN1 enabled for both workers, the completion time drops to 7.5 seconds. By starting a third one we can improve this result to around 5.5 seconds, and Spark is now network-limited.

5.5 Perils of Opportunistic GRIN Usage

There are a number of issues that may be caused by the transition to a GRIN topology. The additional flows may increase buffer pressure and overall latency, while servers could find themselves competing with neighbours for their own uplinks. Our main goal in this regard is to do no worse than the original topology. To deal with these two issues we employ a simple priority scheme, based on DSCP. Each direct flow receives a high-priority code point (such as EF), while secondary flows retain the default low value. We rely on iperf to test the fair use of uplinks, and on a simple client-server program to measure the latency of small transfers. GRIN specific contention may happen in two distinct situations: server-local when multiple flows use the uplink and at the switch, on the egress

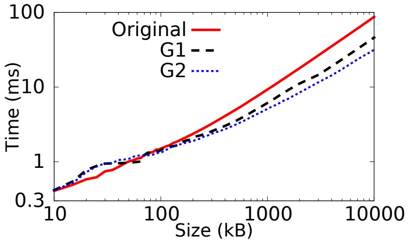


Figure 8: Improvements depend on the size of the transfer.

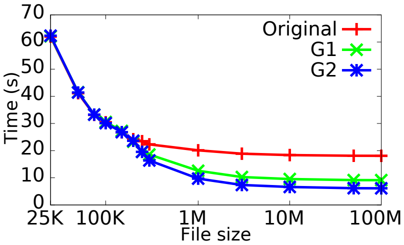


Figure 9: NFS

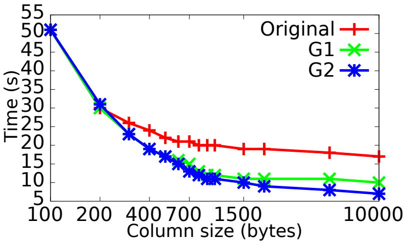


Figure 10: Cassandra

port leading to a particular server. In order to honor DSCP markings locally, servers use a priority-aware queuing discipline, such as PRIO in Linux. Our experiments show that a single, high-priority flow is able to fully utilise the uplink, regardless of the presence of low priority flows.

An idle 10Gbps link takes 100μs on average to transfer 1KB, and 140μs for 100KB with TCP. When competing with several running iperf connections without priorities, the transfer time increases to around 1.8ms in both situations. With the PRIO qdisc, it decreases to 350μs for 1KB and 400μs for 100K. On the downlink, the transfer time grows in both cases to around 2.4ms without priority. If we add priority and configure the switch to discriminate based on DSCP marking, the latency drops to around 110μs and 160μs, respectively.

Does enabling GRIN slow down resource-intensive local applications? Our tests show that GRIN forwarding does not impact storage bound apps, but it is interesting to examine CPU bound and memory bound scenarios.

We ran three resource-intensive applications, Linux kernel compilation, video transcoding and a memcached server, on one of our 6-core Xeon servers. Where relevant, we use a ram disk for persistent storage. Running time is the metric for kernel compilation and transcoding. For memcached, we preload 2^{20} keys with corresponding 60 byte values, and then measure the number of requests that can be fulfilled during 20 second intervals. The requests are generated by 60 local threads that connect to the server using UNIX sockets.

The first two lines from the results in Figure 11 show that running the app on five cores gives near-identical performance regardless of whether the sixth core is idle or forwarding traffic bidirectionally at 10Gbps: if we can spare a core for forwarding, there will be negligible impact on all other apps. The last two lines show the overhead when we run the app on all cores: here forwarding decreases application performance by 9%-13%.

We stress, however, that in many cases clusters of computers are dedicated to one distributed application (e.g. web-search) to avoid bad performance interactions. In such cases the side-effects of forwarding are irrelevant as long as the application as a whole runs faster. In the next section we describe results with GRIN-aware applications where the total completion time is reduced despite the negative effect of forwarding.

5.6 GRIN-aware applications

One simple and very effective optimisation is to schedule bandwidth intensive jobs onto servers that are not direct GRIN neighbors. We have optimised HDFS for reads by placing replicas of the same block in this manner. When a read request comes in, the scheduler replies with the least-recently accessed server that has a replica, and records that the server and his neighbor were "accessed".

We deployed the optimised version of HDFS on 1000 EC2 instances. In each experiment we used a fraction of the nodes to transfer a 400MB file from HDFS to local storage. We used a replication factor of three and a block size equal to 140MB. The results, found in Figure 12, compare the default implementation of HDFS, HDFS running over GRIN1 and, finally, HDFS optimised for GRIN. The results show that running HDFS opportunistically improves download time on average by 14%, and running optimised HDFS brings a 28% improvement. The results also show that, as expected, the optimised version is superior at higher loads, where the probability of GRIN neighbours to be active is much higher.

Next, we optimised Hadoop and Spark by changing the job placement algorithm to avoid scheduling mappers and reducers on neighbour nodes, whenever possible. We ran Spark to run the same string occurrence problem in a large 24GB file over the 1Gbps network. Two HDFS nodes store the data and two Spark nodes do the actual processing. Without GRIN, the total execution time is around 111 seconds. Even if all nodes are grouped up together, using GRIN1 brings the execution time down to 106 seconds. This is caused by a somewhat uneven distribution of the data (one node holds 11G and the other 13G). Since the application can process it pretty fast, GRIN allows one server to help the other out after it finishes sending all the local data. With optimisations enabled, every node has an idle neighbour, and the execution time drops

	kernel	memcache	transcode
idle, 5 cores	137s	5.9 mreq/s	122s
fw, 5 cores	139s	5.9 mreq/s	124s
idle, 6 cores	118s	6m mreq/s	106s
fw, 6 cores	132s	5.5 mreq/s	120s

Figure 11: GRIN forwarding brings little overheads if resource-hungry apps run on separate cores from forwarding. In the worst case, the overhead is 13%

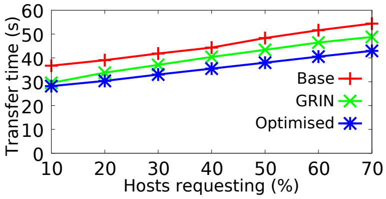


Figure 12: HDFS running on 1000 EC2 instances

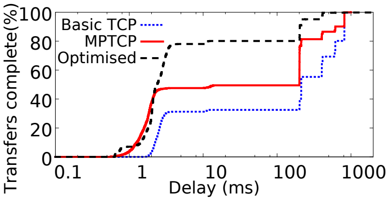


Figure 13: Optimising scatter-gather apps for GRIN

to around 54.5s. A similar optimisation for Hadoop reduces the overall shuffle duration by 20%.

A GRIN topology could also be used to lessen the effects of incast by leveraging the additional switch buffers available at neighbors. In this case, it makes no sense to spread data over multiple paths with MPTCP.

Instead, the frontend can disable MPTCP and split its connections over its uplink and GRIN interfaces in a round-robin fashion. To test this optimisation, we used synthetic scatter-gather application to periodically request data from multiple sources. Figure 13 shows the behavior encountered when one server simultaneously requests 30KB of data over 27 persistent connections, evenly distributed among the nine remaining servers. There are 50 rounds of transfers, each being followed by a 300ms waiting period. Here we compare the original topology with a GRIN2 setup using MPTCP and another one using the incast optimisation. For each case, we plot the CDF of transfer times. Incast mode provides dramatic improvements, reducing the mean by two orders of magnitude.

6 Related Work

A preliminary version of this paper has appeared as [2]. Various solutions have been proposed for increasing core utilisation in full bisection networks, such as MPTCP [17] or Hedera [4]. These, however, aim to make full-bisection topologies behave like a non-blocking switch by routing flows in the network to avoid collisions.

Other approaches, such as Flyways [12] or C-Through [23], are based on augmenting an oversubscribed network with additional communication channels that can be used to improve throughput between different groups of servers when the initial latency is not an issue. They try to create the illusion of full bisection in oversubscribed networks; however, the network activity of a single server is still limited by its uplink.

Using datacenter servers to forward traffic is not a new idea. In fact, topologies such as DCell[11] or BCube[10] rely on servers having multiple ports, and most forwarding is done by the servers themselves. However, these proposals are very difficult to wire, involve complex routing schemes and impose a great forwarding effort on servers. These drawbacks have prevented adoption in practice. GRIN borrows the idea of server routing but uses it in a very simple setup where wiring and routing are trivial, while inherently limiting the amount of forwarded traffic.

Proposals such as ServerSwitch [15] show how forwarding can be implemented in the NIC, without involving the host; such proposals would prevent GRIN forwarding from interfering with host applications.

7 Conclusions

As long as the network is the main bottleneck, wiring up free server ports with GRIN is a simple yet powerful solution to increase the amount of bandwidth available to end-hosts. It is cheap and feasible to implement over almost any topology used today, and this can be done in an incremental fashion. While full-bisection networks show the most potential, we believe it can also be used with oversubscribed networks as long as there still is significant underutilisation.

Even when the additional network ports are not “free”, GRIN can offer an interesting trade-off where we get more capacity out of the network by investing at the edge. There is also the possibility of trading additional cabling costs and complexity to alleviate ToR-level congestion. The server forwarding overhead can be considered an issue in certain opportunistic usage scenarios, but we argue that it can be eliminated altogether with proper hardware support. Moreover, making distributed applications GRIN-aware can significantly diminish any possible shortcoming of the setup. GRIN offers better peak performance, is cheaper, and scales better than alternative solutions like multihoming.

Acknowledgements

This work was supported by Trilogy 2, a research project funded by the European Commission in its Seventh Framework program (FP7 317756). We would like to thank the anonymous reviewers for their feedback and our shepherd George Porter for his help in revising the paper.

References

[1] GNU Linear Programming Kit. `http://www.gnu.org/software/glpk/`.

[2] A. Agache and C. Raiciu. Grin: utilizing the empty half of full bisection networks. In *Proceedings of the 4th USENIX conference on Hot Topics in Cloud Ccomputing*, HotCloud'12, pages 7–7, Berkeley, CA, USA, 2012. USENIX Association.

[3] M. Al-Fares, A. Loukissas, and A. Vahdat. A scalable, commodity data center network architecture. In *Proceedings of ACM SIGCOMM 2008*.

[4] M. Al-Fares, S. Radhakrishnan, B. Raghavan, N. Huang, and A. Vahdat. Hedera: Dynamic flow scheduling for data center networks. In *Proceedings of USENIX NSDI 2010*.

[5] P. Barham, B. Dragovic, K. Fraser, S. H, T. Harris, A. Ho, R. Neugebauer, I. Pratt, and A. Warfield. Xen and the art of virtualization. In *In SOSP (2003*, pages 164–177.

[6] T. Benson, A. Anand, A. Akella, and M. Zhang. Understanding data center traffic characteristics. *SIGCOMM Comput. Commun. Rev.*, 40(1):92–99, Jan. 2010.

[7] M. Carbone and L. Rizzo. Dummynet revisited. *SIGCOMM Comput. Commun. Rev.*, 40(2):12–20, Apr. 2010.

[8] J. Dean and S. Ghemawat. Mapreduce: Simplified data processing on large clusters. *Commun. ACM*, 51(1):107–113, Jan. 2008.

[9] A. Greenberg, N. Jain, S. Kandula, C. Kim, P. Lahiri, D. Maltz, P. Patel, and S. Sengupta. VL2: A scalable and flexible data center network. In *Proceedings of ACM SIGCOMM 2009*.

[10] 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 *Proceedings of ACM SIGCOMM 2009*.

[11] C. Guo, H. Wu, K. Tan, L. Shi, Y. Zhang, and S. Lu. Dcell: a scalable and fault-tolerant network structure for data centers. In *Proceedings of the ACM SIGCOMM 2008 conference on Data communication*, SIGCOMM '08, pages 75–86, New York, NY, USA, 2008. ACM.

[12] D. Halperin, S. Kandula, J. Padhye, P. Bahl and D. Wetherall. Augmenting data center networks with multi-gigabit wireless links. In *Proceedings of ACM SIGCOMM 2011*.

[13] S. Kandula, S. Sengupta, A. Greenberg, P. Patel, and R. Chaiken. The nature of data center traffic: measurements & analysis. In *Proceedings of ACM IMC 2009*.

[14] A. Lakshman and P. Malik. Cassandra: a decentralized structured storage system. *SIGOPS Oper. Syst. Rev.*, 44(2):35–40, Apr. 2010.

[15] G. Lu, C. Guo, Y. Li, Z. Zhou, T. Yuan, H. Wu, Y. Xiong, R. Gao, and Y. Zhang. Serverswitch: a programmable and high performance platform for data center networks. In *Proceedings of the 8th USENIX conference on Networked systems design and implementation*, NSDI'11, pages 2–2, Berkeley, CA, USA, 2011. USENIX Association.

[16] B. Pawlowski, D. Noveck, D. Robinson, and R. Thurlow. The nfs version 4 protocol. In *In Proceedings of the 2nd International System Administration and Networking Conference (SANE 2000*, 2000.

[17] C. Raiciu, S. Barre, C. Pluntke, A. Greenhalgh, D. Wischik, and M. Handley. Improving datacenter performance and robustness with multipath TCP. In *Proceedings of ACM SIGCOMM 2011*.

[18] C. Raiciu, M. Ionescu, and D. Niculescu. Opening up black box networks with CloudTalk. In *Proceedings of USENIX HotCloud 2012*.

[19] C. Raiciu, C. Paasch, S. Barre, A. Ford, M. Honda, F. Duchene, O. Bonaventure, and M. Handley. How hard can it be? designing and implementing a deployable multipath tcp. In *Proceedings of the 9th USENIX Conference on Networked Systems Design and Implementation*, NSDI'12, pages 29–29, Berkeley, CA, USA, 2012. USENIX Association.

[20] L. Rizzo. netmap: a novel framework for fast packet I/O. In *Proceedings of USENIX ATC 2012*.

[21] K. Shvachko, H. Kuang, S. Radia, and R. Chansler. The hadoop distributed file system. In *Proceedings of the 2010 IEEE 26th Symposium on Mass Storage Systems and Technologies (MSST)*, MSST '10, pages 1–10, Washington, DC, USA, 2010. IEEE Computer Society.

[22] 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 Comput. Commun. Rev.*, 39(4):303–314, Aug. 2009.

[23] G. Wang, D. G. Andersen, M. Kaminsky, K. Papagiãnnaki, T. E. Ng, M. Kozuch, and M. Ryan. c-Through: Part-time optics in data centers. In *Proceedings of ACM SIGCOMM 2010*.

[24] D. Wischik, C. Raiciu, A. Greenhalgh, and M. Handley. Design, implementation and evaluation of congestion control for multipath TCP. In *Proceedings of USENIX NSDI 2011*.

[25] M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauley, M. J. Franklin, S. Shenker, and I. Stoica. Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. In *Proceedings of the 9th USENIX Conference on Networked Systems Design and Implementation*, NSDI'12, pages 2–2, Berkeley, CA, USA, 2012. USENIX Association.

[26] M. Zaharia, A. Konwinski, A. D. Joseph, R. Katz, and I. Stoica. Improving mapreduce performance in heterogeneous environments. In *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation*, OSDI'08, pages 29–42, Berkeley, CA, USA, 2008. USENIX Association.

Designing Distributed Systems Using Approximate Synchrony in Data Center Networks

Dan R. K. Ports   Jialin Li   Vincent Liu   Naveen Kr. Sharma   Arvind Krishnamurthy
University of Washington
{drkp,lijl,vincent,naveenks,arvind}@cs.washington.edu

Abstract

Distributed systems are traditionally designed independently from the underlying network, making worst-case assumptions (*e.g.,* complete asynchrony) about its behavior. However, many of today's distributed applications are deployed in data centers, where the network is more reliable, predictable, and extensible. In these environments, it is possible to co-design distributed systems with their network layer, and doing so can offer substantial benefits.

This paper explores network-level mechanisms for providing *Mostly-Ordered Multicast (MOM)*: a best-effort ordering property for concurrent multicast operations. Using this primitive, we design *Speculative Paxos*, a state machine replication protocol that relies on the network to order requests in the normal case. This approach leads to substantial performance benefits: under realistic data center conditions, Speculative Paxos can provide 40% lower latency and 2.6× higher throughput than the standard Paxos protocol. It offers lower latency than a latency-optimized protocol (Fast Paxos) with the same throughput as a throughput-optimized protocol (batching).

1 Introduction

Most distributed systems are designed independently from the underlying network. For example, distributed algorithms are typically designed assuming an asynchronous network, where messages may be arbitrarily delayed, dropped, or reordered in transit. In order to avoid making assumptions about the network, designers are in effect making worst-case assumptions about it.

Such an approach is well-suited for the Internet, where little is known about the network: one cannot predict what paths messages might take or what might happen to them along the way. However, many of today's applications are distributed systems that are deployed in data centers. Data center networks have a number of desirable properties that distinguish them from the Internet:

- Data center networks are more *predictable*. They are designed using structured topologies [8,15,33], which makes it easier to understand packet routes and expected latencies.
- Data center networks are more *reliable*. Congestion losses can be made unlikely using features such as Quality of Service and Data Center Bridging [18].
- Data center networks are more *extensible*. They are part of a single administrative domain. Combined with new flexibility provided by modern technologies like software-defined networking, this makes it possible to deploy new types of in-network processing or routing.

These differences have the potential to change the way distributed systems are designed. It is now possible to co-design distributed systems and the network they use, building systems that rely on stronger guarantees available in the network and deploying new network-level primitives that benefit higher layers.

In this paper, we explore the benefits of co-designing in the context of state machine replication—a performance-critical component at the heart of many critical data center services. Our approach is to treat the data center as an approximation of a synchronous network, in contrast to the asynchronous model of the Internet. We introduce two new mechanisms, a new network-level primitive called *Mostly-Ordered Multicast* and the *Speculative Paxos* replication protocol, which leverages approximate synchrony to provide higher performance in data centers.

The first half of our approach is to engineer the network to provide stronger ordering guarantees. We introduce a *Mostly-Ordered Multicast* primitive (MOM), which provides a best-effort guarantee that all receivers will receive messages from different senders in a consistent order. We develop simple but effective techniques for providing Mostly-Ordered Multicast that leverage the structured topology of a data center network and the forwarding flexibility provided by software-defined networking.

Building on this MOM primitive is *Speculative Paxos*, a new protocol for state machine replication designed for an environment where reordering is rare. In the normal case, Speculative Paxos relies on MOM's ordering guarantees to efficiently sequence requests, allowing it to execute and commit client operations with the minimum possible latency (two message delays) and with significantly higher throughput than Paxos. However, Speculative Paxos remains correct even in the uncommon case where messages are delivered out of order: it falls back on a reconciliation protocol that ensures it remains safe and live with the same guarantees as Paxos.

Our experiments demonstrate the effectiveness of this approach. We find:

- Our customized network-level multicast mechanism ensures that multicast messages can be delivered in a consistent order with greater than 99.9% probability in a data center environment.
- In these environments, Speculative Paxos provides 40% lower latency and 2.6× higher throughput than leader-based Paxos.
- Speculative Paxos can provide the best of both worlds: it offers 20% lower latency than a latency-optimized protocol (Fast Paxos) while providing the same throughput as a throughput-optimized protocol (batching).

We have used Speculative Paxos to implement various components of a transactional replicated key-value store. Compared to other Paxos variants, Speculative Paxos allows this application to commit significantly more transactions within a fixed latency budget.

2 Background

Our goal is to implement more efficient replicated services by taking advantage of features of the data center environment. To motivate our approach, we briefly discuss existing consensus algorithms as well as the design of typical data center networks upon which they are built.

2.1 Replication and Consensus Algorithms

Replication is widely used to provide highly available and consistent services in data centers. For example, services like Chubby [4] and ZooKeeper [17] provide applications with support for distributed coordination and synchronization. Similarly, persistent storage systems like H-Store [39], Granola [10], and Spanner [9] require multiple replicas to commit updates. This provides better availability than using a single replica and also provides better performance by eschewing costly synchronous disk writes in favor of maintaining multiple copies in RAM.

Replication systems rely on a consensus protocol (*e.g.,* Paxos [24, 25], Viewstamped Replication [29, 35], or atomic broadcast [3, 19]) to ensure that operations execute in a consistent order across replicas. In this paper, we consider systems that provide a *state machine replication* interface [23, 38]. Here, a set of nodes are either *clients* or *replicas*, which both run application code and interact with each other using the replication protocol. Note that, here, clients are application servers in the data center, not end-users. Clients submit *requests* containing an operation to be executed. This begins a multi-round protocol to ensure that replicas agree on a consistent ordering of operations before executing the request.

As an example, consider the canonical state machine replication protocol, leader-based Paxos. In the normal case, when there are no failures, requests are processed as shown in Figure 1. One of the replicas is designated as the *leader*, and is responsible for ordering requests; the leader can be replaced if it fails. Clients submit requests to the leader. The leader assigns each incoming request a *sequence number*, and sends a PREPARE message to the other replicas containing the request and sequence number. The other replicas record the request in the log and acknowledge with a PREPARE-OK message. Once the leader has received responses from enough replicas, it executes the operation and replies to the client.

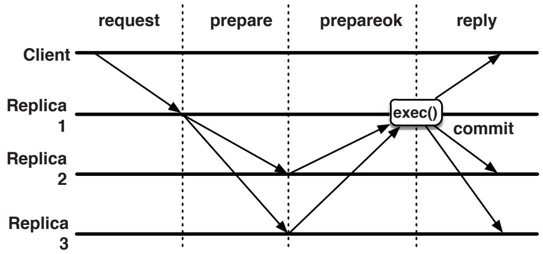


Figure 1: Normal-case execution of Multi-Paxos/Viewstamped Replication

Data center applications demand high performance from replicated systems. These systems must be able to execute operations with both high throughput and low latency. The latter is an increasingly important factor for modern web applications that routinely access data from thousands of storage servers, while needing to keep the total latency within strict bounds for interactive applications [37]. For replication protocols, throughput is typically a function of the load on a bottleneck entity, *e.g.,* the leader in Figure 1, which processes a disproportionate number of messages. Latency is primarily a function of the number of message delays in the protocol—for Paxos, four message delays from when a client submits its request until it receives a reply.

2.2 Data Centers

Today's data centers incorporate highly engineered networks to provide high availability, high throughput, low latency, and low cost. Operators take advantage of the following properties to tune their networks:

Centralized control. All of the infrastructure is in a single administrative domain, making it possible for operators to deploy large-scale changes. Software-defined networking tools (*e.g.,* OpenFlow) make it possible to implement customized forwarding rules coordinated by a central controller.

A structured network. Data center networks are multi-rooted trees of switches typically organized into three levels. The leaves of the tree are Top-of-Rack (ToR) switches that connect down to many machines in a rack, with a rack containing few tens of servers. These ToR switches are interconnected using additional switches or routers, which are organized into an *aggregation tier* in the middle and a *core tier* at the top. Each ToR switch is connected to multiple switches at the next level, thus providing desired resilience in the face of link or switch failures. Racks themselves are typically grouped into a *cluster* (about ten to twenty racks) such that all connectivity within a cluster is provided by just the bottom two levels of the network.

Within the data center, there may be many replicated services: for example, Google's Spanner and similar storage systems use one replica group per shard, with hundreds or thousands of shards in the data center. The replicas in each group will be located in different racks (for failure-independence) but may be located in the same cluster to simplify cluster management and scheduling. The service will receive requests from clients throughout the data center.

Switch support for QoS. The controlled setting also makes it possible to deploy services that can transmit certain types of messages (*e.g.,* control messages) with higher priority than the rest of the data center traffic. These priorities are implemented by providing multiple hardware or software output queues—one for each priority level. When using strict priorities, the switch will always pull from higher priority queues before lower priority queues. The length and drop policy of each queue can be tuned to drop lower priority traffic first and can also be tuned to minimize latency jitter.

3 Mostly-Ordered Multicast

The consensus algorithms described in the previous section rely heavily on the concept of ordering. Most Paxos deployments dedicate a leader node to this purpose; approaches such as Fast Paxos [27] rely on requests to arrive in order. We argue instead that the structured, highly-engineered networks used in data centers can themselves be used to order operations in the normal case. To that end, this section explores different network-layer options for providing a *mostly-ordered multicast (MOM)* mechanism. We show that simple techniques can effectively provide best-effort ordering in a data center.

3.1 Model

We consider multicast primitives that allow clients to communicate simultaneously with a group of receivers N.

In this category, the traditional *totally-ordered multicast* provides the following property: if $n_i \in N$ processes a multicast message m followed by another multicast message m', then any other node $n_j \in N$ that receives m' must process m before m'. Primitives like this are common in group communication systems [3]. Ensuring that this property holds even in the presence of failures is a problem equivalent to consensus, and would obviate the need for application code to run protocols like Paxos at all.

Instead, we consider a relaxed version of this property, which does not require it to hold in every case. A multicast implementation is said to possess the *mostly-ordered multicast* property if the above ordering constraint is satisfied with high frequency. This permits occasional *ordering violations*: these occur if n_i processes m followed by m' and either (1) n_j processes m after m', or (2) n_j does not process m at all (because the message is lost).

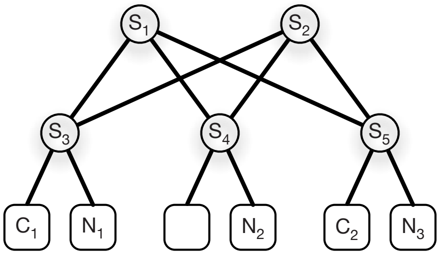


Figure 2: Clients C_1 and C_2 communicating to a multicast group comprising of N_1, N_2, and N_3.

This is an empirical property about the common-case behavior, not a strict guarantee. As a result, MOMs can be implemented as a best-effort network primitive. We seek to take advantage of the properties of the data center network previously described in order to implement MOMs efficiently in the normal case. The property may be violated in the event of transient network failures or congestion, so application-level code must be able to handle occasional ordering violations.

In this section, we first examine why existing multicast mechanisms do not provide this property, and then describe three techniques for implementing MOMs. Each stems from the idea of equalizing path length between multicast messages with a topology-aware multicast. The second adds QoS techniques to equalize latency while the third leverages in-network serialization to guarantee correct ordering. In Section 3.4, we evaluate these protocols using both an implementation on an OpenFlow switch testbed and a simulation of a datacenter-scale network. We show that the first two techniques are effective at providing MOMs with a reordering rate of ~0.01–0.1% and the third eliminates reorderings entirely except during network failures.

3.2 Existing Multicast is Not Ordered

We first consider existing network-layer multicast mechanisms to understand why ordering violations occur.

Using IP multicast, a client can send a single multicast message to the target multicast address and have it delivered to all of the nodes. Multicast-enabled switches will, by default, flood multicast traffic to all the ports in a broadcast domain. Unnecessary flooding costs can be eliminated by using IGMP, which manages the membership of a multicast group.

Using a network-level multicast mechanism, packets from different senders may be received in conflicting orders because they traverse paths of varying lengths and experience varying levels of congestion along different links. For example, suppose the clients C_1 and C_2 in Figure 2 each send a multicast message to the multicast group $\{N_1, N_2, N_3\}$ at times $t = 0$ and $t = \varepsilon$ respectively. Let $p_{i\to j}$ represent the network path traversed by the multicast operation initiated by C_i to N_j, and $l(p_{i\to j})$ represent its latency. In this setting, the ordering property is violated if N_1 receives m_1 followed by m_2 while N_3 receives m_2 followed by m_1, which could occur if $l(p_{1\to 1}) < l(p_{2\to 1}) + \varepsilon$ and $l(p_{1\to 3}) > l(p_{2\to 3}) + \varepsilon$. This is distinctly possible since the paths $p_{1\to 1}$ and $p_{2\to 3}$ traverse just two links each while $p_{2\to 1}$ and $p_{1\to 3}$ traverse four links each.

In practice, many applications do not even use network-level multicast; they use application-level multicast mechanisms such as having the client send individual unicast messages to each of the nodes in the target multicast group. This approach, which requires no support from the network architecture, has even worse ordering properties. In addition to the path length variation seen in network-level multicast, there is additional latency skew caused by the messages not being sent at the same time.

3.3 Our Designs

We can improve the ordering provided by network-level multicast by building our own multicast mechanisms. Specifically, we present a sequence of design options that provide successively stronger ordering guarantees.

1. Topology-aware multicast: Ensure that all multicast messages traverse the same number of links. This eliminates reordering due to path dilation.
2. High-priority multicast: Use topology-aware multicast, but also assign high QoS priorities to multicasts. This essentially eliminates drops due to congestion, and also reduces reordering due to queuing delays.
3. In-network serialization: Use high-priority multicast, but route all packets through a single root switch. This eliminates all remaining non-failure related reordering.

The common intuition behind all of our designs is that messages can be sent along predictable paths through the data center network topology with low latency and high reliability in the common case.

We have implemented these three designs using OpenFlow [31] software-defined networking, which allows it to be deployed on a variety of switches. OpenFlow provides access to switch support for custom forwarding rules, multicast replication, and even header rewriting. Our designs assume the existence of a SDN controller for ease of configuration and failure recovery. The controller installs appropriate rules for multicast forwarding, and updates them when switch failures are detected.

3.3.1 Topology-Aware Multicast

In our first design, we attempt to minimize the disparity in message latencies by assuring that the messages corresponding to a single multicast operation traverse the same number of links. To achieve this, we route multicast messages through switches that are equidistant from all of the nodes in the target multicast group. The routes are dependent on the scope of the multicast group.

For instance, if all multicast group members are located within the same cluster in a three-level tree topology, the aggregation switches of that cluster represent the nearest set of switches that are equidistant from all members. For datacenter-wide multicast groups, multicast messages are routed through the root switches.

Equalizing path lengths can cause increased latency for some recipients of each multicast message, because messages no longer take the shortest path to each recipient. However, the maximum latency—and, in many cases, the average latency—is not significantly impacted: in datacenter-wide multicast groups, some messages would have to be routed through the root switches anyway. Moreover, the cost of traversing an extra link is small in data center networks, particularly in comparison to the Internet. As a result, this tradeoff is a good one: Speculative Paxos is able to take advantage of the more predictable ordering to provide better end-to-end latency.

Addressing. Each multicast group has a single address. In the intra-cluster case, the address shares the same prefix as the rest of the cluster, but has a distinct prefix from any of the ToR switches. In the datacenter-wide case, the address should come from a subnet not used by any other cluster.

Routing. The above addressing scheme ensures that, with longest-prefix matching, multicast messages are routed to the correct set of switches without any changes to the existing routing protocols. The target switches will each have specific rules that convert the message to a true multicast packet and will send it downward along any replica-facing ports. Lower switches will replicate the multicast packet on multiple ports as necessary.

Note that in this scheme, core switches have a multicast rule for every datacenter-wide multicast group while aggregation switches have a rule for every multicast group within its cluster. Typical switches have support for thousands to tens of thousands of these multicast groups.

As an example of end-to-end routing, suppose the client C_1 in Figure 2 wishes to send a message to a three-member multicast group that is spread across the data center. The group's multicast address will be of a subnet not shared by any cluster, and thus will be routed to either S_1 or S_2. Those core switches will then replicate it into three messages that are sent on each of the downward links. This simple mechanism guarantees that all messages traverse the same number of links.

Failure Recovery. When a link or switch fails, the SDN controller will eventually detect the failure and route around it as part of the normal failover process. In many cases, when there is sufficient path redundancy inside the network (as with a data center network organized into a Clos topology [15]), this is sufficient to repair MOM routing as well. However, in some cases—most notably a traditional fat tree, where there is only a single path from each root switch down to a given host—some root switches may become unusable for certain replicas. In these cases, the controller installs rules in the network that "blacklist" these root switches for applicable multicast groups. Note, however, that failures along upward links in a fat tree network can be handled locally by simply redirecting MOM traffic to any working root without involving the controller [30].

3.3.2 High-Priority Multicast

The above protocol equalizes path length and, in an unloaded network, significantly reduces the reordering rate of multicast messages. However, in the presence of background traffic, different paths may have different queuing delays. For example, suppose clients C_1 and C_2 in Figure 2 send multicasts m_1 and m_2 through S_1 and S_2 respectively. The latency over these two switches might vary significantly. If there is significant cross-traffic over the links $S_1 - S_5$ and $S_2 - S_4$ but not over the links $S_1 - S_4$ and $S_2 - S_5$, then N_2 is likely to receive m_1 followed by m_2 while N_3 would receive them in opposite order.

We can easily mitigate the impact of cross-traffic on latency by assigning a higher priority to MOM traffic using Quality of Service (QoS) mechanisms. Prioritizing this traffic is possible because this traffic is typically a small fraction of overall data center traffic volume. By assigning MOM messages to a strict-priority hardware queue, we can ensure that MOM traffic is always sent before other types of traffic. This limits the queuing delays introduced by cross-traffic to the duration of a single packet. With 10 Gbps links and 1500 byte packets, this corresponds to a worst-case queuing delay of about 1.2 μs per link or a total of about 2.4 μs from a core switch to a ToR switch. Our evaluation results show that this leads to a negligible amount of reordering under typical operating conditions.

3.3.3 In-Network Serialization

Even with QoS, currently-transmitting packets cannot be preempted. This fact, combined with minor variations in switch latency imply that there is still a small chance of message reordering. For these cases, we present an approach that uses the network itself to *guarantee* correct ordering of messages in spite of cross-traffic.

Our approach is to route all multicast operations to a given group through the same switch. This top-level switch not only serves as a serialization point, but also ensures that messages to a given group node traverse the same path. As a result, it provides perfect ordering as long as the switch delivers packets to output ports in order (we have not observed this to be a problem, as discussed in Section 3.4.1) and there are no failures.

Addressing/Routing. As before, we assign a single address to each multicast group. In this design, however, the SDN controller will unilaterally designate a root or aggregation switch as a serialization point and install the appropriate routes in the network. By default, we hash the multicast addresses across the relevant target switches for load balancing and ease of routing.

In certain network architectures, similar routing functionality can also be achieved using PIM Sparse Mode multicast [12], which routes multicast packets from all sources through a single "rendezvous point" router.

Failure Recovery. Like the failure recovery mechanism of Section 3.3.1, we can also rely on an SDN controller to route around failures. If a switch no longer has a valid path to all replicas of a multicast group or is unreachable by a set of clients, the controller will remap the multicast group's address to a different switch that *does* have a valid path. This may require the addition of routing table entries across a handful of switches in the network. These rules will be more specific than the default, hashed mappings and will therefore take precedence. Downtime due to failures is minimal: devices typically have four or five 9's of reliability [14] and path recovery takes a few milliseconds [30].

We can further reduce the failure recovery time by letting the end hosts handle failover. We can do this by setting up n different multicast serialization points each with their own multicast address and pre-installed routing table entries. By specifying the multicast address, the clients thus choose which serialization point to use. A client can failover from one designated root switch to another immediately after it receives switch failure notifications from the fabric controller or upon encountering a persistent communication failure to a target MOM group. This approach requires some additional application-level complexity and increases routing table/address usage, but provides much faster failover than the baseline.

Load Balancing. In-network serialization is not inherently load-balancing in the same way as our previous designs: all multicast traffic for a particular group traverses the same root switch. However, as described previously, there are many replica groups, each with message load far below the capacity of a data center core switch. Different groups will be hashed to different serialization switches, providing load balancing in aggregate. If necessary, the SDN controller can explicitly specify the root switch for particular groups to achieve better load balancing.

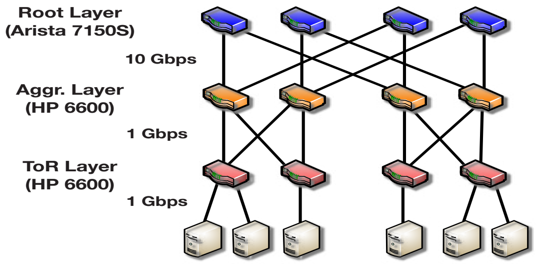


Figure 3: Testbed configuration

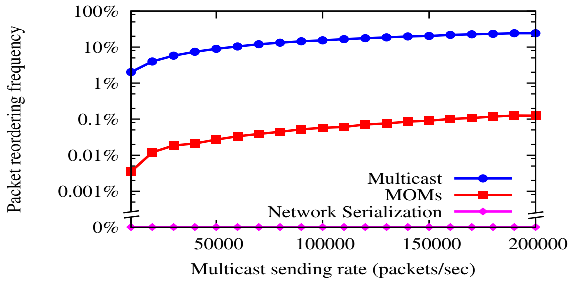


Figure 4: Measured packet reorder rates on 12-switch testbed

3.4 Evaluation of MOMs

Can data center networks effectively provide mostly-ordered multicast? To answer this question, we conducted a series of experiments to determine the reorder rate of concurrent multicast transmissions. We perform experiments using a multi-switch testbed, and conduct simulations to explore larger data center deployments.

3.4.1 Testbed Evaluation

We evaluate the ordering properties of our multicast mechanism using a testbed that emulates twelve switches in a fat-tree configuration, as depicted in Figure 3. The testbed emulates four Top-of-Rack switches in two clusters, with two aggregation switches per cluster and four root switches connecting the clusters. Host-ToR and ToR-aggregation links are 1 Gbps, while aggregation-root links are 10 Gbps. This testbed captures many essential properties of a data center network, including path length variance and the possibility for multicast packets to arrive out of order due to multi-path effects. The testbed can deliver multicast messages either using native IP multicast, topology-aware MOM, or network serialization.

The testbed is realized using VLANs on five switches. Four HP ProCurve 6600 switches implement the ToR and aggregation switches, and an Arista 7150S-24 10 Gbps switch implements the root switches. All hosts are Dell PowerEdge R610 servers with 4 6-core Intel Xeon L5640 CPUs running Ubuntu 12.04, using Broadcom BCM5709 1000BaseT adapters to connect to the ToRs.

A preliminary question is whether individual switches will cause concurrent multicast traffic to be delivered in conflicting orders, which could occur because of parallelism in switch processing. We tested this by connecting multiple senders and receivers to the same switch, and verified that all receivers received multicast traffic in the same order. We did not observe any reorderings on any of the switches we tested, including the two models in our testbed, even at link-saturating rates of multicast traffic.

With the testbed configured as in Figure 3, we connected three senders and three receivers to the ToR switches. The receivers record the order of arriving multicasts and compare them to compute the frequency of ordering violations.

In this configuration, ordering violations can occur because multicast packets traverse different paths between switches. Figure 4 shows that this occurs frequently for conventional IP multicast, with as many as 25% of packets reordered. By equalizing path lengths, our topology-aware MOM mechanism reduces the reordering frequency by 2–3 orders of magnitude. Network serialization eliminates reorderings entirely.

3.4.2 Simulation Results

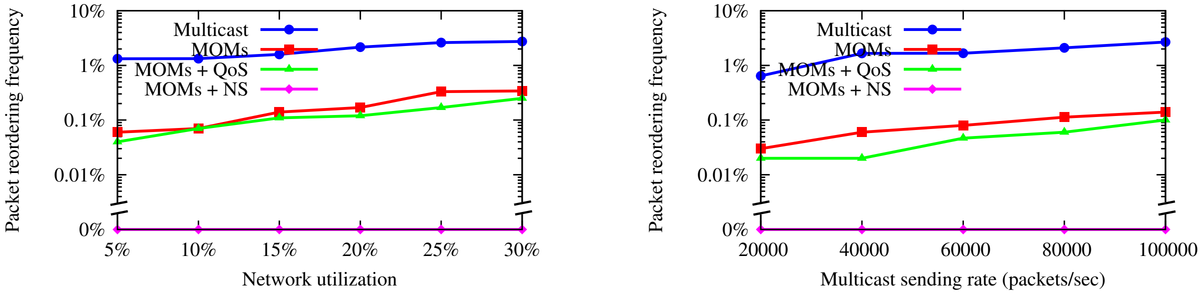
To evaluate the effectiveness of our proposed multicast designs on a larger network, we use a parallel, packet-level, event-driven simulator. In addition to the experiments described below, we have validated the simulator by simulating the same topology as our testbed, using measured latency distributions for the two switch types. The simulated and measured results match to within 8%.

The simulated data center network topology consists of 2560 servers and a total of 119 switches. The switches are configured in a three-level FatTree topology [1] with a total oversubscription ratio of about 1:4. The core and aggregation switches each have 16 10 Gbps ports while the ToRs each have 8 10 Gbps ports and 40 1 Gbps ports. Each switch has a strict-priority queue in addition to standard queues. Both queues use drop-tail behavior and a switching latency distribution taken from our measurements of the Arista 7150 switch.

In this setup, we configure end hosts to periodically transmit MOM traffic to a multicast group of 3 nodes where we observe the frequency of ordering violations. In addition to MOM traffic, hosts also send background traffic to one another. We derive the distribution of interarrival times and ON/OFF-period length for the background traffic from measurements of Microsoft data centers [2].

In the experiments in Figure 5, we measure reordering rates for the four options previously discussed: standard multicast, topology-aware multicast (MOMs), MOMs with QoS, and MOMs with in-network serialization.

Reordering. In Figure 5(a), we fix the MOM sending rate to 50,000 messages per second and vary the amount of simulated background traffic. The range equates to about 5–30% average utilization. Note that data center traffic is extremely bursty, so this range is typical [2]. In



(a) Reorder rates with varying network utilization; MOM sending rate fixed at 50,000 per second.

(b) Reorder rates with varying MOM throughput; background traffic fixed at 10% average utilization.

Figure 5: Simulated packet reorder rates, in a 160-switch, three-level fat-tree network

Figure 5(b), we vary the MOM sending rate and fix the average utilization to 10%. Results are similar for other utilization rates.

As expected, the standard multicast approach has a relatively high rate of packet reorderings because packets traverse paths of varying lengths. Simply being aware of the topology reduces the rate of reorderings by an order of magnitude, and employing QoS prioritization mitigates the impact of congestion caused by other traffic. The in-network serialization approach achieves perfect ordering: as packets are routed through a single switch, only congestion losses could cause ordering violations.

Latency. As we previously observed, MOM can increase the path length of a multicast message to the longest path from a sender to one of the receivers. As a result, the time until a message arrives at the first receiver increases. However, for more than 70% of messages, the *average* latency over all receivers remains unchanged. The latency skew, i.e., the difference between maximum and minimum delivery time to all recipients for any given message, is in all cases under 2.5 μs for the in-network serialization approach.

4 Speculative Paxos

Our evaluation in the previous section shows that we can engineer a data center network to provide MOMs. How should this capability influence our design of distributed systems? We argue that a data center network with MOMs can be viewed as *approximately synchronous*: it provides strong ordering properties in the common case, but they may occasionally be violated during failures.

To take advantage of this model, we introduce *Speculative Paxos*, a new state machine replication protocol. Speculative Paxos relies on MOMs to be ordered in the common case. Each replica speculatively executes requests based on this order, before agreement is reached. This speculative approach allows the protocol to run with the minimum possible latency (two message delays) and provides high throughput by avoiding communication between replicas on each request. When occasional ordering violations occur, it invokes a reconciliation protocol to rollback inconsistent operations and agree on a new state. Thus, Speculative Paxos does not rely on MOM for correctness, only for efficiency.

Client interface

- `invoke`(*operation*) → *result*

Replica interface

- `speculativelyExecute`(*seqno*, *operation*) → *result*
- `rollback`(*from-seqno*, *to-seqno*, *list*<*operations*>)
- `commit`(*seqno*)

Figure 6: Speculative Paxos library API

4.1 Model

Speculative Paxos provides state machine replication, following the model in Section 2.1. In particular, it guarantees linearizability [16] provided that there are no more than *f* failures: operations appear as though they were executed in a consistent sequential order, and each operation sees the effect of operations that completed before it. The API of the Speculative Paxos library is shown in Figure 6.

Speculative Paxos differs from traditional state machine replication protocols in that it executes operations speculatively at the replicas, *before* agreement is reached about the ordering of requests. When the replica receives a request, the Speculative Paxos library makes a `speculativelyExecute` upcall to the application code, providing it with the requested operation and an associated sequence number. In the event of a failed speculation, the Speculative Paxos library may make a `rollback` upcall, requesting that the application undo the most recent operations and return to a previous state. To do so, it provides the sequence number and operation of all the commands to be rolled back. The Speculative Paxos library also periodically makes `commit` upcalls to indicate that previously-speculative operations will never be rolled back, allowing the application to discard information (*e.g.,* undo logs) needed to roll back operations.

Importantly, although Speculative Paxos executes operations speculatively on replicas, *speculative state is not exposed to clients*. The Speculative Paxos library only returns results to the client application after they are known to have successfully committed in the same order at a quorum of replicas. In this respect, Speculative Paxos is similar to Zyzzyva [22], and differs from systems that employ speculation on the client side [41].

Failure Model Although the premise for our work is that data center networks can provide stronger guarantees of ordering than distributed algorithms typically assume, Speculative Paxos does not rely on this assumption for correctness. It remains correct under the same assumptions as Paxos and Viewstamped Replication: it requires $2f+1$ replicas and provides safety as long as no more than f replicas fail simultaneously, even if the network drops, delays, reorders, or duplicates messages. It provides liveness as long as messages that are repeatedly resent are eventually delivered before the recipients time out. (This requirement is the same as in Paxos, and is required because of the impossibility of consensus in an asynchronous system [13].)

4.2 Protocol

Speculative Paxos consists of three sub-protocols:

- *Speculative processing* commits requests efficiently in the normal case where messages are ordered and $< f/2$ replicas have failed (Section 4.2.2)
- *Synchronization* periodically verifies that replicas have speculatively executed the same requests in the same order (Section 4.2.3)
- *Reconciliation* ensures progress when requests are delivered out of order or when between $f/2$ and f nodes have failed (Section 4.2.4)

4.2.1 Replica State

Each replica maintains a *status*, a *log*, and a *view number*.

The replica's status indicates whether it can process new operations. Most of the time, the replica is in the NORMAL state, which allows speculative processing of new operations. While the reconciliation protocol is in progress, the replica is in the RECONCILIATION state. There are also RECOVERY and RECONFIGURATION states used when a failed replica is reconstructing its state and when the membership of the replica set is changing.

The log is a sequence of operations executed by the replica. Each entry in the log is tagged with a *sequence number* and a state, which is either COMMITTED or SPECULATIVE. All committed operations precede all speculative operations. Each log entry also has an associated *summary hash*. The summary hash of entry n is given by

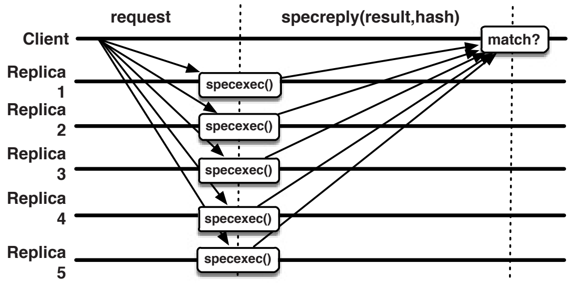


Figure 7: Speculative processing protocol.

$$summary_n = H(summary_{n-1} \,||\, operation_n)$$

Thus, it summarizes the replica's state up to that point: two replicas that have the same summary hash for log entry n must agree on the order of operations up to entry n. To simplify exposition, we assume replicas retain their entire logs indefinitely; a standard checkpointing procedure can be used to truncate logs.

The system moves through a series of views, each with a designated view number and leader. Each replica maintains its own idea of the current view number. The leader is selected in a round-robin ordering based on the view number, *i.e.,* for view v, replica v mod n is the leader. This is similar to leader election in [6] and [29], but the leader does not play a special role in the normal-case speculative processing protocol. It is used only to coordinate synchronization and reconciliation.

4.2.2 Speculative Processing

Speculative Paxos processes requests speculatively in the common case. When a client application initiates an operation, the Speculative Paxos library sends a REQUEST message to all replicas. This message includes the operation requested, the identity of a client, and a unique per-client request identifier. The REQUEST message is sent using our MOM primitive, ensuring that replicas are likely to receive concurrent requests in the same order.

Replicas participate in speculative processing when they are in the NORMAL state. Upon receiving a REQUEST message, they immediately speculatively execute the request: they assign the request the next higher sequence number, append the request to the log in the SPECULATIVE state, and make an upcall to application code to execute the request. It then sends a SPECULATIVE-REPLY message to the client, which includes the result of executing the operation as well as the sequence number assigned to the request and a summary hash of the replica's log.

Clients wait for SPECULATIVE-REPLY messages from a *superquorum* of $f + \lceil f/2 \rceil + 1$ replicas, and compare the responses. If all responses match exactly, *i.e.,* they have the same sequence number and summary hash, the client treats the operation as committed. The matching responses indicate that the superquorum of replicas have executed the request (and all previous operations) in the same order. The replicas themselves do not yet know that the operation has committed, but the reconciliation protocol ensures that any such operation will persist even if there are failures, and may not be rolled back. If the client fails to receive SPECULATIVE-REPLY messages from a superquorum of replicas before a timeout, or if the responses do not match (indicating that the replicas are not in the same state), it initiates a reconciliation, as described in Section 4.2.4.

Why is a superquorum of responses needed rather than a simple majority as in Paxos? The reasoning is the same as in Fast Paxos[1]: even correct replicas may receive operations from different clients in inconsistent orders. Consider what would happen if we had used a quorum of size $f+1$, and one request was executed by $f+1$ replicas and a different request by the other f. If any of the replicas in the majority subsequently fail, the recovery protocol will not be able to distinguish them. As a result, the size a superquorum must be chosen such that, if an operation is successful, a majority of active replicas will have that operation in their log [28].

4.2.3 Synchronization

In the speculative processing protocol, clients learn that their requests have succeeded when they receive matching speculative replies from a superquorum of replicas. The replicas, however, do not communicate with each other as part of speculative processing, so they do not learn the outcome of the operations they have executed. The synchronization protocol is a periodic process, driven by the leader, that verifies that replicas are in the same state.

Periodically (every t milliseconds, or every k requests), the leader initiates synchronization by sending a $\langle$SYNC, v, $s\rangle$ message to the other replicas, where v is its current view number and s is the highest sequence number in its log. Replicas respond to SYNC messages by sending the leader a message $\langle$SYNC-REPLY, v, s, $h(s)\rangle$, where $h(s)$ is the summary hash associated with entry s in its log. If v is not the current view number, or the replica is not in the NORMAL state, the message is ignored.

When the leader has received $f + \lceil f/2 \rceil + 1$ SYNC-REPLY messages for a sequence number s, including its own, it checks whether the hashes in the messages match. If so, the replicas agree on the ordering of requests up to s. The leader promotes all requests with sequence number less than or equal to s from SPECULATIVE to COMMITTED state, and makes a `commit(`s`)` upcall to the application. It then sends a message $\langle$COMMIT, v, s, $h(s)\rangle$ to the other replicas. Replicas receiving this message also commit all operations up to s if their summary hash matches.

If the leader receives SYNC-REPLY messages that do not have the same hash, or if a replica receives a COMMIT message with a different hash than its current log entry, it initiates a reconciliation.

4.2.4 Reconciliation

From time to time, replica state may diverge. This can occur if messages are dropped or reordered by the network, or if replicas fail. The reconciliation protocol repairs divergent state and ensures that the system makes progress.

The reconciliation protocol follows the same general structure as view changes in Viewstamped Replication [29]: all replicas stop processing new requests and send their log to the new leader, which selects a definitive log and distributes it to the other replicas. The main difference is that the leader must perform a more complex *log merging* procedure that retains any operation that successfully completed even though operations may have been executed in different orders at different replicas.

When a replica begins a reconciliation, it increments its view number and sets its status to RECONCILIATION, which stops normal processing of client requests. It then sends a $\langle$START-RECONCILIATION, $v\rangle$ message to the other replicas. The other replicas, upon receiving a START-RECONCILIATION message for a higher view than the one they are currently in, perform the same procedure. Once a replica has received START-RECONCILIATION messages for view v from f other replicas, it sends a $\langle$RECONCILE, v, v_ℓ, $log\rangle$ message to the leader of the new view. Here, v_ℓ is the last view in which the replica's status was NORMAL.

Once the new leader receives RECONCILE messages from f other replicas, it merges their logs. The log merging procedure considers all logs with the highest v_ℓ (including the leader's own log, if applicable) and produces a combined log with two properties:

- If the same prefix appears in a majority of the logs, then those entries appear in the combined log in the same position.
- Every operation in any of the input logs appears in the output log.

The first property is critical for correctness: it ensures that any operation that might have successfully completed at a client survives into the new view. Because clients treat requests as successful once they have received matching summary hashes from $f + \lceil f/2 \rceil + 1$ replicas, and f of those replicas might have subsequently failed, any successful operation will appear in at least $\lceil f/2 \rceil + 1$ logs.

The second property is not required for safety, but ensures that the system makes progress. Even if all multicast requests are reordered by the network, the reconciliation

[1] Fast Paxos is typically presented as requiring quorums of size $2f+1$ out of $3f+1$, but like Speculative Paxos can be configured such that $2f+1$ replicas can make progress with f failures but need superquorums to execute fast rounds [27].

procedure selects a definitive ordering of all requests.

The procedure for merging logs is as follows:

- The leader considers only logs with the highest v_ℓ; any other logs are discarded. This ensures that the results of a previous reconciliation are respected.
- It then selects the log with the most COMMITTED entries. These operations are known to have succeeded, so they are added to the combined log.
- Starting with the next sequence number, it checks whether a majority of the logs have an entry with the same summary hash for that sequence number. If so, that operation is added to the log in SPECULATIVE state. This process is repeated with each subsequent sequence number until no match is found.
- It then gathers all other operations found in any log that have not yet been added to the combined log, selects an arbitrary ordering for them, and appends them to the log in the SPECULATIVE state.

The leader then sends a $\langle$START-VIEW, v, $log\rangle$ message to all replicas. Upon receiving this message, the replica *installs* the new log: it rolls back any speculative operations in its log that do not match the new log, and executes any new operations in ascending order. It then sets its current view to v, and resets its status to NORMAL, resuming speculative processing of new requests.

Ensuring progress with f failures. The reconciliation protocol uses a quorum size of $f+1$ replicas, unlike the speculative processing protocol, which requires a superquorum of $f+\lceil f/2\rceil+1$ replicas. This means that reconciliation can succeed even when more than $f/2$ but no more than f replicas are faulty, while speculative processing cannot. Because reconciliation ensures that all operations submitted before the reconciliation began are assigned a consistent order, it can commit operations even if up to f replicas are faulty.

Upon receiving a START-VIEW message, each replica also sends a $\langle$IN-VIEW, $v\rangle$ message to the leader to acknowledge that it has received the log for the new view. Once the leader has received IN-VIEW messages from f other replicas, it commits all of the speculative operations that were included in the START-VIEW message, notifies the clients, and notifies the other replicas with a COMMIT message. This allows operations to be committed even if there are more than $f/2$ failed replicas. This process is analogous to combining regular and fast rounds in Fast Paxos: only $f+1$ replicas are required in this case because only the leader is is allowed to propose the ordering of requests that starts the new view.

4.2.5 Recovery and Reconfiguration

Replicas that have failed and rejoined the system follow a *recovery* protocol to ensure that they have the current state. A *reconfiguration* protocol can also be used to change the membership of the replica group, *e.g.,* to replace failed replicas with new ones. For this purpose, Speculative Paxos uses standard recovery and reconfiguration protocols from Viewstamped Replication [29]. The reconfiguration protocol also includes the need to add or remove newly-joined or departing replicas to the multicast group. For our OpenFlow multicast forwarding prototype, it requires contacting the OpenFlow controller.

Reconfiguration can also be used to change the system from Speculative Paxos to a traditional implementation of Paxos or VR. Because reconfiguration can succeed with up to f failures, this can be a useful strategy when more than $f/2$ failures occur, or during transient network failures that can cause packets to be reordered.

4.3 Correctness

Speculative Paxos treats an operation as successful (and notifies the client application) if the operation is COMMITTED in at least one replica's log, or if it is SPECULATIVE in a common prefix of $f+\lceil f/2\rceil+1$ replica's logs.

Any successful operation always survives in the same serial order. We only need to consider reconciliations here, as speculative processing will only add new operations to the end of the log, and synchronization will cause successful operations to be COMMITTED. Consider first operations that succeeded because they were speculatively executed on $f+\lceil f/2\rceil+1$ replicas. These operations will survive reconciliations. The reconciliation process requires $f+1$ out of $2f+1$ replicas to respond, so $\lceil f/2\rceil+1$ logs containing these operations will be considered, and the log merging algorithm ensures they will survive in the same position.

Operations can also be committed through reconciliation. This can happen only once $f+1$ replicas have processed the START-VIEW message for that view. All of these replicas agree on the ordering of these operations, and at least one will participate in the next reconciliation, because $f+1$ replicas are required for reconciliation. The reconciliation procedure only merges logs with the highest v_ℓ, and the log merging procedure will ensure that the common prefix of these logs survives.

Only one operation can succeed for a given sequence number. By itself, the speculative processing protocol allows only one operation to succeed for a given sequence number, because an operation only succeeds if speculatively committed by a superquorum of replicas. The reconciliation protocol will not assign a different operation to any sequence number that could potentially have speculatively committed at enough replicas, nor one that committed as the result of a previous reconciliation.

	Latency (Msg Delays)	Message Complexity	Messages at Bottleneck Replica
Paxos	4	$2n$	$2n$
Paxos + batching	4+	$2+\frac{2n}{b}$	$2+\frac{2n}{b}$
Fast Paxos	3	$2n$	$2n$
Speculative Paxos	2	$2n+\frac{2n}{s}$	$2+\frac{2n}{s}$

Table 1: Comparison of Paxos, Fast Paxos, and Speculative Paxos. n is the total number of replicas; b is the batch size for Paxos with batching, and s is the number of requests between synchronization for Speculative Paxos.

4.4 Discussion

Speculative Paxos offers high performance because it commits most operations via the fast-path speculative execution protocol. It improves on the latency of client operations, an increasingly critical factor in today's applications: a client can submit a request and learn its outcome in two message delays—the optimal latency, and a significant improvement over the four message delays of leader-based Paxos, as shown in Table 1. Speculative Paxos also provides better throughput, because it has no bottleneck replica that bears a disproportionate amount of the load. In Speculative Paxos, each replica processes only two messages (plus periodic synchronizations), whereas all $2n$ messages are processed by the leader in Paxos.

Speculative Paxos is closely related to Fast Paxos [27], which reduces latency by sending requests directly from clients to all replicas. Fast Paxos also incurs a penalty when different requests are received by the replicas in a conflicting order. In Fast Paxos, the message flow is

client → replicas → leader → client

and so the protocol requires three message delays. Speculative Paxos improves on this by executing operations speculatively so that clients can learn the result of their operations in two message delays. The tradeoff is that the reconciliation protocol is more expensive and speculative operations might need to be rolled back, making Speculative Paxos slower in conflict-heavy environments. This tradeoff is an example of our co-design philosophy: Fast Paxos would also benefit from MOM, but Speculative Paxos is optimized specially for an environment where multicasts are mostly ordered.

Speculative Paxos improves throughput by reducing the number of messages processed by each node. Despite the name, Fast Paxos does not improve throughput, although it reduces latency: the leader still processes $2n$ messages, so it remains a bottleneck. Other variants on Paxos aim to reduce this bottleneck. A common approach is to batch requests at the leader, only running the full protocol periodically. This also eliminates a bottleneck, increasing throughput dramatically, but *increases* rather than reducing latency.

4.5 Evaluation

We have implemented the Speculative Paxos protocol as a library for clients and replicas. Our library comprises about 10,000 lines of C++, and also supports leader-based Paxos (with or without batching) and Fast Paxos.

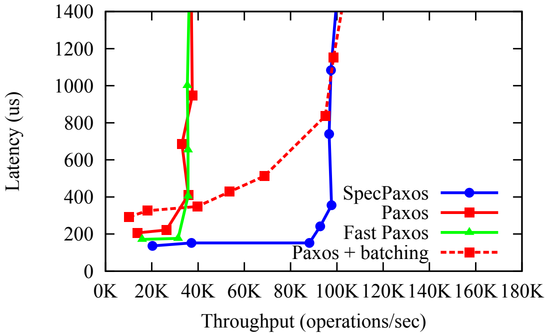


Figure 8: Latency vs throughput tradeoff for testbed deployment

We evaluate the performance of Speculative Paxos and compare it to Paxos and Fast Paxos using a deployment on the twelve-switch testbed shown in Figure 3, measuring the performance tradeoffs under varying client load. We then investigate the protocol's sensitivity to network conditions by emulating MOM ordering violations.

4.5.1 Latency/Throughput Comparison

In our testbed experiments, we use three replicas, so $f=1$, and a superquorum of all three replicas is required for Speculative Paxos or Fast Paxos to commit operations on the fast path. The replicas and multiple client hosts are connected to the ToR switches with 1 Gbps links. Speculative Paxos and Fast Paxos clients use the network serialization variant of MOM for communicating with the replicas; Paxos uses standard IP multicast.

Figure 8 plots the median latency experienced by clients against the overall request throughput obtained by varying the number of closed-loop clients from 2 to 300. We compare Speculative Paxos, Fast Paxos, and Paxos with and without batching. In the batching variant, we use the latency-optimized sliding-window batching strategy from PBFT [6], with batch sizes up to 64. (This limit on batch sizes is not reached in this environment; higher maximum batch sizes have no effect.) This comparison shows:

- At low to medium request rates, Speculative Paxos provides lower latency (135 μs) than either Paxos (220 μs) or Fast Paxos (171 μs). This improvement can be attributed to the fewer message delays.

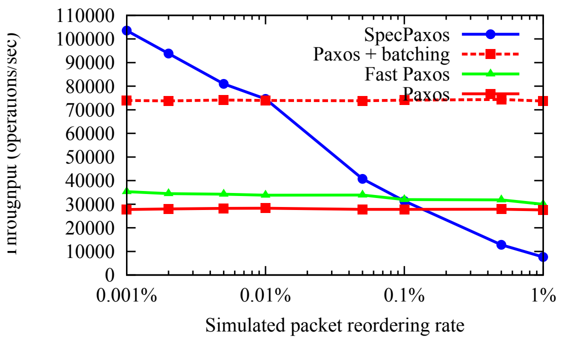


Figure 9: Throughput with simulated packet reordering

Application	Total LoC	Rollback LoC
Timestamp Server	154	10
Lock Manager	606	75
Key-value Store	2011	248

Table 2: Complexity of rollback in test applications

- Speculative Paxos is able to sustain a higher throughput level (∼100,000 req/s) than either Paxos or Fast Paxos (∼38,000 req/s), because fewer messages are handled by the leader, which otherwise becomes a bottleneck.
- Like Speculative Paxos, batching also increases the throughput of Paxos substantially by eliminating the leader bottleneck: the two achieve equivalent peak throughput levels. However, batching also increases latency: at a throughput level of 90,000 req/s, its latency is 3.5 times higher than that of Speculative Paxos.

4.5.2 Reordering Sensitivity

To gain further insight into Speculative Paxos performance under varying conditions, we modified our implementation to artificially reorder random incoming packets. These tests used three nodes with Xeon L5640 processors connected via a single 1 Gbps switch.

We measured throughput with 20 concurrent closed-loop clients. When packet reordering is rare, Speculative Paxos outperforms Paxos and Fast Paxos by a factor of 3×, as shown in Figure 9. As the reorder rate increases, Speculative Paxos must perform reconciliations and performance drops. However, it continues to outperform Paxos until the reordering rate exceeds 0.1%. Our experiments in Section 3.4 indicate that data center environments will have lower reorder rates using topology-aware multicast, and can eliminate orderings entirely except in rare failure cases using network serialization. Paxos performance is largely unaffected by reordering, and Fast Paxos throughput drops slightly because conflicts must be resolved by the leader, but this is cheaper than reconciliation.

5 Applications

We demonstrate the benefits and tradeoffs involved in using speculation by implementing and evaluating several applications ranging from the trivial to fairly complex.

Since Speculative Paxos exposes speculation at the application replicas, it requires rollback support from the application. The application must be able to rollback operations in the event of failed speculations.

We next describe three applications ranging from a simple timestamp server to a complex transactional, distributed key-value store inspired by Spanner [9]. We measure the performance achieved by using Speculative Paxos and comment on the complexity of rollback code.

Timestamp Server. This network service generates monotonically increasing timestamps or globally unique identifier numbers. Such services are often used for distributed concurrency control. Each replica maintains its own counter which is incremented upon a new request. On rollback, the counter is simply decremented once for each request to be reverted.

Lock Manager. The Lock Manager is a fault-tolerant synchronization service which provides a fine-grained locking interface. Clients can acquire and release locks in read or write mode. Each replica maintains a mapping of object locks held by a client and the converse. On rollback, both these mappings are updated by the inverse operation, *e.g.,* RELEASE(X) for LOCK(X). Since these operations do not commute, they must be rolled back in the reverse order in which they were applied.

Transactional Key-Value Store. We built a distributed in-memory key-value store which supports serializable transactions using two-phase commit and strict two-phase locking or optimistic concurrency control (OCC) to provide concurrency control. Client can perform GET and PUT operations, and commit them atomically using BEGIN, COMMIT, and ABORT operations.

The key-value store keeps multiple versions of each row (like many databases) to support reading a consistent snapshot of past data, and to implement optimistic concurrency control. Rolling back PUT operations requires reverting a key to an earlier version, and rolling back PREPARE, COMMIT, and ABORT two-phase commit operations requires adjusting transaction metadata to an earlier state.

We test the key-value store on the previously-described testbed. In our microbenchmark, a single client executes transactions in a closed loop. Each transaction involves several replicated get and put operations. Speculative Paxos commits a transaction in an average of 1.01 ms, a 30% improvement over the 1.44 ms required for Paxos, and a 10% improvement over the 1.12 ms for Fast Paxos.

We also evaluate the key-value store on a more complex benchmark: a synthetic workload based on a profile of the Retwis open-source Twitter clone. The workload chooses

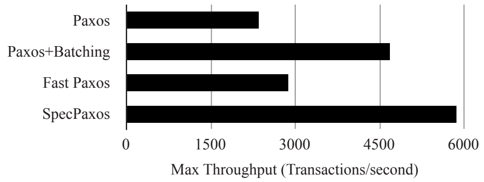


Figure 10: Maximum throughput attained by key-value store within 10 ms SLO

keys based on a Zipf distribution, and operations based on the transactions implemented in Retwis. Figure 10 plots the maximum throughput the system can achieve while remaining within a 10 ms SLO. By simultaneously providing lower latency and eliminating throughput bottlenecks, Speculative Paxos achieves significantly greater throughput within this latency budget.

6 Related Work

Weak Synchrony. The theoretical distributed systems literature has studied several models of weak synchrony assumptions. These include bounding the latency of message delivery and the relative speeds of processors [11], or introducing unreliable failure detectors [7]. In particular, a single Ethernet LAN segment has been shown to be nearly synchronous in practice, where even short timeouts are effective [40] and reorderings are rare [21]. The data center network is far more complex; we have shown that with existing multicast mechanisms, reorderings are frequent, but our network-level MOM mechanisms can be used to ensure ordering with high probability.

In the context of a single LAN, the Optimistic Atomic Broadcast [36] protocol provides total ordering of messages under the assumption of a spontaneous total ordering property equivalent to our MOM property, and was later used to implement a transaction processing system [21]. Besides extending this idea to the more complex data center context, the Speculative Paxos protocol is more heavily optimized for lower reorder rates. For example, the OAB protocol requires all-to-all communication; Speculative Paxos achieves higher throughput by avoiding this. Speculative Paxos also introduces additional mechanisms such as summary hashes to support a client/server state machine replication model instead of atomic broadcast.

Paxos Variants. Speculative Paxos is similar to Fast Paxos [27], which reduces latency when messages arrive at replicas in the same order. Speculative Paxos takes this approach further, eliminating another message round and communication between the replicas, in exchange for reduced performance when messages are reordered.

Total ordering of operations is not always needed. Generalized Paxos [26] and variants such as Multicoordinated Paxos [5] and Egalitarian Paxos [32] mitigate the cost of conflicts in a Fast Paxos-like protocol by requiring the programmer to identify requests that commute and permitting such requests to commit in different orders on different replicas. Such an approach could allow Speculative Paxos to tolerate higher reordering rates.

Speculation. Speculative Paxos is also closely related to recent work on speculative Byzantine fault tolerant replication. The most similar is Zyzzyva [22], which employs speculation on the server side to reduce the cost of operations when replicas are non-faulty. Like Speculative Paxos, Zyzzyva replicas execute requests speculatively and do not learn the outcome of their request, but clients can determine if replicas are in a consistent state. Zyzzyva's speculation assumes that requests are not assigned conflicting orders by a Byzantine primary replica. Speculative Paxos applies the same idea in a non-Byzantine setting, speculatively assuming that requests are not reordered by the network. Eve [20] uses speculation and rollback to allow non-conflicting operations to execute concurrently; some of its techniques could be applied here to reduce the cost of rollback.

An alternate approach is to apply speculation on the *client* side. SpecBFT [41] modifies the PBFT protocol so that the primary sends an immediate response to the client, which continues executing speculatively until it later receives confirmation from the other replicas. This approach also allows the client to resume executing after two message delays. However, the client cannot communicate over the network or issue further state machine operations until the speculative state is committed. This significantly limits its usability for data center applications that often perform a sequence of accesses to different storage systems [37]. Client-side speculation also requires kernel modifications to support unmodified applications [34].

7 Conclusion

We have presented two mechanisms: the Speculative Paxos protocol, which achieves higher performance when the network provides our Mostly-Ordered Multicast property, and new network-level multicast mechanisms designed to provide this ordering property. Applied together with MOM, Speculative Paxos achieves significantly higher performance than standard protocols in data center environments. This example demonstrates the benefits of co-designing a distributed system with its underlying network, an approach we encourage developers of future data center applications to consider.

Acknowledgements

We would like to thank Adriana Szekeres, Irene Zhang, our shepherd Robbert van Renesse, and the anonymous reviewers for their feedback. This work was supported in part by Google and the National Science Foundation (CNS-0963754, CNS-1217597, CNS-1318396, and CNS-1420703).

References

[1] M. Al-Fares, A. Loukissas, and A. Vahdat. A scalable, commodity data center network architecture. In *Proceedings of ACM SIGCOMM 2008*, Seattle, WA, USA, Aug. 2008.

[2] T. Benson, A. Akella, and D. A. Maltz. Network traffic characteristics of data centers in the wild. In *Proceedings of the 10th ACM SIGCOMM Conference on Internet Measurement (IMC '10)*, Nov. 2010.

[3] K. P. Birman and T. A. Joseph. Exploiting virtual synchrony in distributed systems. In *Proceedings of the 11th ACM Symposium on Operating Systems Principles (SOSP '87)*, Austin, TX, USA, Oct. 1987.

[4] M. Burrows. The Chubby lock service for loosely-coupled distributed systems. In *Proceedings of the 7th USENIX Symposium on Operating Systems Design and Implementation (OSDI '06)*, Seattle, WA, USA, Nov. 2006.

[5] L. Camargos, R. Schmidt, and F. Pedone. Multicoordinated Paxos. Technical report, University of Lugano Faculty of Informatics, 2007/02, Jan. 2007.

[6] M. Castro and B. Liskov. Practical Byzantine fault tolerance. In *Proceedings of the 3rd USENIX Symposium on Operating Systems Design and Implementation (OSDI '99)*, New Orleans, LA, USA, Feb. 1999.

[7] T. D. Chandra, V. Hadzilacos, and S. Toueg. The weakest failure detector for solving consensus. *Journal of the ACM*, 43(4):685–722, July 1996.

[8] Cisco data center infrastructure design guide 2.5. https://www.cisco.com/application/pdf/en/us/guest/netsol/ns107/c649/ccmigration_09186a008073377d.pdf.

[9] J. C. Corbett, J. Dean, M. Epstein, A. Fikes, C. Frost, J. Furman, S. Ghemawat, A. Gubarev, C. Heiser, P. Hochschild, W. Hsieh, S. Kanthak, E. Kogan, H. Li, A. Lloyd, S. Melnik, D. Mwaura, D. Nagle, S. Quinlan, R. Rao, L. Rolig, Y. Saito, M. Szymaniak, C. Taylor, R. Wang, and D. Woodford. Spanner: Google's globally-distributed database. In *Proceedings of the 10th USENIX Symposium on Operating Systems Design and Implementation (OSDI '12)*, Hollywood, CA, USA, Oct. 2012.

[10] J. Cowling and B. Liskov. Granola: Low-overhead distributed transaction coordination. In *Proceedings of the 2012 USENIX Annual Technical Conference*, Boston, MA, USA, June 2012.

[11] C. Dwork, N. Lynch, and L. Stockmeyer. Consensus in the presence of partial synchrony. *Journal of the ACM*, 35(2):228–323, Apr. 1988.

[12] D. Estrin, D. Farinacci, A. Helmy, D. Thaler, S. Deering, M. Handley, V. Jacobson, C. Liu, P. Sharma, and L. Wei. Protocol independent multicast-sparse mode (PIM-SM): Protocol specification. RFC 2117, June 1997. https://tools.ietf.org/html/rfc2117.

[13] M. J. Fischer, N. A. Lynch, and M. S. Patterson. Impossibility of distributed consensus with one faulty process. *Journal of the ACM*, 32(2):374–382, Apr. 1985.

[14] P. Gill, N. Jain, and N. Nagappan. Understanding network failures in data centers: Measurement, analysis, and implications. In *Proceedings of ACM SIGCOMM 2011*, Toronto, ON, Canada, Aug. 2011.

[15] A. Greenberg, J. R. Hamilton, 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 *Proceedings of ACM SIGCOMM 2009*, Barcelona, Spain, Aug. 2009.

[16] M. P. Herlihy and J. M. Wing. Linearizabiliy: A correctness condition for concurrent objects. *ACM Transactions on Programming Languages and Systems*, 12(3):463–492, July 1990.

[17] P. Hunt, M. Konar, F. P. Junqueira, and B. Reed. ZooKeeper: Wait-free coordination for Internet-scale systems. In *Proceedings of the 2010 USENIX Annual Technical Conference*, Boston, MA, USA, June 2010.

[18] IEEE 802.1 Data Center Bridging. http://www.ieee802.org/1/pages/dcbridges.html.

[19] F. Junqueira, B. Reed, and M. Serafini. Zab: High-performance broadcast for primary-backup systems. In *Proceedings of the 41st IEEE/IFIP International Conference on Dependable Systems and Networks (DSN '11)*, Hong Kong, China, June 2011.

[20] M. Kapritsos, Y. Wang, V. Quema, A. Clement, L. Alvisi, and M. Dahlin. All about Eve: Execute-verify replication for multi-core servers. In *Proceedings of the 10th USENIX Symposium on Operating Systems Design and Implementation (OSDI '12)*, Hollywood, CA, USA, Oct. 2012.

[21] B. Kemme, F. Pedone, G. Alonso, and A. Schiper. Processing transactions over optimistic atomic broadcast protocols. In *Proceedings of the 13th International Symposium on Distributed Computing (DISC '99)*, Bratislava, Slovakia, Sept. 1999.

[22] R. Kotla, L. Alvisi, M. Dahlin, A. Clement, and E. Wong. Zyzzyva: Speculative Byzantine fault tolerance. In *Proceedings of the 21th ACM Symposium on Operating Systems Principles (SOSP '07)*, Stevenson, WA, USA, Oct. 2007.

[23] L. Lamport. Time, clocks, and ordering of events in a distributed system. *Communications of the ACM*, 21(7):558–565, July 1978.

[24] L. Lamport. The part-time parliament. *ACM Transactions on Computer Systems*, 16(2):133–169, May 1998.

[25] L. Lamport. Paxos made simple. *ACM SIGACT News*, 32(4):18–25, Dec. 2001.

[26] L. Lamport. Generalized consensus and Paxos. Technical Report MSR-TR-2005-33, Microsoft Research, Mar. 2005.

[27] L. Lamport. Fast Paxos. *Distributed Computing*, 19(2):79–103, Oct. 2006.

[28] L. Lamport. Lower bounds for asynchronous consensus. *Distributed Computing*, 19(2):104–125, Oct. 2006.

[29] B. Liskov and J. Cowling. Viewstamped replication revisited. Technical Report MIT-CSAIL-TR-2012-021, MIT Computer Science and Artificial Intelligence Laboratory, Cambridge, MA, USA, July 2012.

[30] V. Liu, D. Halperin, A. Krishnamurthy, and T. Anderson. F10: A fault-tolerant engineered network. In *Proceedings of the 10th USENIX Symposium on Networked Systems Design and Implementation (NSDI '13)*, Lombard, IL, USA, Apr. 2013.

[31] N. McKeown, T. Anderson, H. Balakrishnan, G. Parulkar, L. Peterson, J. Rexford, S. Shenker, and J. Turner. OpenFlow: enabling innovation in campus networks. *ACM SIGCOMM Computer Communication Review*, 38(2):69–74, Apr. 2008.

[32] I. Moraru, D. G. Andersen, and M. Kaminsky. There is more consensus in egalitarian parliaments. In *Proceedings of the 25rd ACM Symposium on Operating Systems Principles (SOSP '13)*, Farmington, PA, USA, Nov. 2013.

[33] R. N. Mysore, A. Pamboris, N. Farrington, N. Huang, P. Miri, S. Radhakrishnan, V. Subramanya, and A. Vahdat. PortLand: A scalable fault-tolerant layer 2 data center network fabric. In *Proceedings of ACM SIGCOMM 2009*, Barcelona, Spain, Aug. 2009.

[34] E. B. Nightingale, P. M. Chen, and J. Flinn. Speculative execution in a distributed file system. In *Proceedings of the 20th ACM Symposium on Operating Systems Principles (SOSP '05)*, Brighton, United Kingdom, Oct. 2005.

[35] B. M. Oki and B. H. Liskov. Viewstamped replication: A new primary copy method to support highly-available distributed systems. In *Proceedings of the 7th ACM Symposium on Principles of Distributed Computing (PODC '88)*, Toronto, Ontario, Canada, Aug. 1988.

[36] F. Pedone and A. Schiper. Optimistic atomic broadcast. In *Proceedings of the 12th International Symposium on Distributed Computing (DISC '98)*, Andros, Greece, Sept. 1998.

[37] S. M. Rumble, D. Ongaro, R. Stutsman, M. Rosenblum, and J. K. Ousterhout. It's time for low latency. In *Proceedings of the 13th Workshop on Hot Topics in Operating Systems (HotOS '11)*, Napa, CA, USA, May 2011.

[38] F. B. Schneider. Implementing fault-tolerant services using the state machine approach: a tutorial. *ACM Computing Surveys*, 22(4):299–319, Dec. 1990.

[39] M. Stonebraker, S. Madden, D. J. Abadi, S. Harizopoulos, N. Hachem, and P. Helland. The end of an architectural era: (it's time for a complete rewrite). In *Proceedings of the 33rd International Conference on Very Large Data Bases (VLDB '07)*, Vienna, Austria, Sept. 2007.

[40] P. Urbán, X. Défago, and A. Schiper. Chasing the FLP impossibility result in a LAN: or, how robust can a fault tolerant server be? In *Proceedings of the 20th IEEE Symposium on Reliable Distributed Systems (SRDS '01)*, New Orleans, LA USA, Oct. 2001.

[41] B. Wester, J. Cowling, E. Nightingale, P. M. Chen, J. Flinn, and B. Liskov. Tolerating latency in replicated state machines through client speculation. In *Proceedings of the 6th USENIX Symposium on Networked Systems Design and Implementation (NSDI '09)*, Boston, MA, USA, Apr. 2009.

Kinetic: Verifiable Dynamic Network Control

Hyojoon Kim[‡], Joshua Reich[*], Arpit Gupta[†]
Muhammad Shahbaz[†], Nick Feamster[†], Russ Clark[‡]
[‡]*Georgia Tech* *AT&T Labs – Research* †*Princeton University*
http://kinetic.noise.gatech.edu/

Abstract

Network conditions are dynamic; unfortunately, current approaches to configuring networks are not. Network operators need tools to express how a network's data-plane behavior should respond to a wide range of events and changing conditions, ranging from unexpected failures to shifting traffic patterns to planned maintenance. Yet, to update the network configuration today, operators typically rely on a combination of manual intervention and ad hoc scripts. In this paper, we present Kinetic, a domain specific language and network control system that enables operators to control their networks dynamically in a concise, intuitive way. Kinetic also automatically verifies the correctness of these control programs with respect to user-specified temporal properties. Our user study of Kinetic with several hundred network operators demonstrates that Kinetic is intuitive and usable, and our performance evaluation shows that realistic Kinetic programs scale well with the number of policies and the size of the network.

1 Introduction

Network conditions are always changing. Traffic patterns change, hosts arrive and depart, topologies change, intrusions occur, and so forth. Despite the fact that many of these changes are predictable—and, in some cases, even planned—an operator's *control* over the network remains relatively static. In response to changing conditions, network operators typically manually change low-level network configurations. Our previous study of network configuration changes found that a campus network may experience anywhere from 1,000 to 18,000 changes per month [20]. Although tools like Puppet [27] and Chef [3] can automate some network device configuration tasks, this level of automation is still relatively hands-on and error-prone.

To underscore the importance of this problem, we analyzed acceptable use policies from more than 20 campus networks (many of which are publicly available [22]) and also surveyed network operators about their experience with existing tools for implementing these kinds of policies. These policies are written in English and typically express how the network's forwarding behavior should change in response to changing network conditions. For example, the University of Illinois's network use policy has an unrestricted class, and four restricted classes of traffic shaping; a user's traffic is downgraded into different classes based on their past usage over a 24-hour sliding window. Such policies sound simple enough when expressed in prose, but in fact they require complex instrumentation and "wrappers" that dynamically change low-level network configuration. Network operators currently have no concise way to express these functions, nor do they have any way of checking whether their changes will result in the intended behavior. In a recent survey we conducted that included several hundred network operators, 89% of respondents said that they could not be certain that the changes they made to network configuration would not introduce new bugs.

Software-defined networking (SDN) is a powerful approach to managing computer networks [11] because it provides network-wide visibility of and control over a network's behavior; the Frenetic [13] family of languages provides higher-level abstractions for expressing network control. These languages are embedded in general-purpose programming languages (specifically, OCaml and Python), which makes it possible to write control programs that can respond to arbitrary events. Yet these languages do not provide intuitive abstractions for *automating* changes to network policy in response to dynamic conditions, nor do they make it possible to *verify* that these changes will match the operator's requirements for how network behavior should react to changing network conditions.

To address these problems, we present Kinetic, a domain specific language (DSL) and SDN controller that enables writing network control programs that capture responses to changing network conditions in a concise, intuitive, and verifiable language. Kinetic provides a structured language for expressing a network policy in terms of finite state machines (FSMs), which both concisely capture dynamics and are amenable to verification. States correspond to distinct forwarding behavior, and events trigger transitions between states. Kinetic's event handler listens to events and triggers transitions in policy, which in turn update the data plane.

Kinetic makes it possible to *verify* that changes to network behavior conform to a higher-level specification of correctness. For example, a network operator might want to prove that a control program would never allow a host access to certain parts of the network once an intrusion has been detected. Ongoing work has devoted much attention to verification of the network's data plane; tools such as VeriFlow [19] and HSA [18] can determine, for example, whether the forwarding table entries in a network's switches and routers would result in persistent loops or reachability problems. However, *these tools only operate on a snapshot of the data plane*; they do not allow operators to reason about network *control programs*, or how network control would change in response to various events or changes in network conditions. They do not provide any way for a network operator to find errors in the control programs that install erroneous data-plane state in the first place. Kinetic's focus on automating and verifying the control plane is complementary to this previous work. Kinetic's use of computation tree logic (CTL) [6]—and its ability to automatically verify policies with the NuSMV model checker [4]— can allow network operators to verify the dynamic behavior of the controller before the control programs are ever run.

One significant challenge we faced when designing Kinetic is the potential for state explosion in Kinetic programs, due to the large number of hosts, flows, network events, and policies. A naïve encoding of dynamic policies in an FSM would result in an exponential number of states, even for simple programs because every flow, with all possible combination of fields (*e.g.*, src/dst IP, src/dst MAC, etc), can have its own state. To control this state explosion, Kinetic introduces an abstraction called a *Located Packet Equivalence Class (LPEC)*, through which a programmer can specify a division of the flow space and map an independent copy of an FSM (FSM instance) to each class of flow space. Using LPECs, a programmer can define groups of flows that should always map to same FSM instances (*e.g.*, all flows from the same source MAC address). Thus, each defined group of flows will be in the same state. Additionally, because Kinetic is itself based on Pyretic (a Python-based SDN control language in the Frenetic family) [25], Kinetic inherits Pyretic's language and runtime features. Specifically, Kinetic uses Pyretic's composition operators to express larger FSMs as multiple smaller ones that correspond to distinct network tasks (*e.g.*, authentication, intrusion detection, rate-limiting). Applying Pyretic's composition operators to independent Kinetic FSMs and classic product construction of automata [10] (combining multiple FSMs with union or product) greatly simplifies the construction of Kinetic's FSM expressions and allows the FSM-based policies to scale.

We evaluated two aspects of Kinetic: (1) its usability, in terms of both conciseness and operators' facility with expressing realistic network policies; and (2) its performance, in terms of its ability to efficiently compile network policies into flow-table entries, particularly as the number of policies, the size of the network, and the rate of events grow. We conducted a user study with Kinetic of more than 650 participants, many of whom were network operators with no prior programming experience; most found Kinetic quite accessible: 79% thought that configuring the network with Kinetic was easier than current approaches, and 84% thought that Kinetic makes it easier to verify network configuration than existing alternatives.

Profession		**Experience (years)**		**# Users in Network**	
Operator	216	1	32	1–10	156
Developer	251	1–5	310	10–100	137
Student	123	5–10	187	100–1,000	136
Vendor	80	10–15	150	1,000–10,000	118
Manager	69	15–20	122	> 10,000	322
Other	138	> 20	73		
Total	877		874		869

Table 1: *Demographics of participants in the Kinetic user study. We asked these participants about their experiences configuring existing networks, as well as their experiences using Kinetic. Section 5 discusses the participants' experience with Kinetic. Not all participants answered every question.*

Kinetic is open-source and publicly available; the project webpage provides access to the source code, a tutorial on Kinetic, and all of the code for the experimental evaluation [21]. The system has been used by SDN practitioners [14] and has served as the basis for projects and assignments in several university courses, as well as in a Coursera [8] course, where it has been used by thousands of students over the past two years.

2 Motivation and Background

To motivate the need for Kinetic, we present the results of a survey of network operators about problems automating and verifying network configuration. We then present background on Pyretic, the language on which Kinetic is based; and on model checking and computation tree logic, which we use to design Kinetic's verification engine.

2.1 Motivation: Network Operator Survey

To gain a better understanding of the extent to which network operators have to change their network configurations, as well as their level of confidence in their changes, we conducted an institutional review board (IRB)-approved survey of more than 800 participants, concerning their experience with configuring existing networks, as part of a Coursera course on software-defined networking that we offer [8]. Table 1 summarizes the demographics of the participants: about 870 students completed the survey, 216 of whom were full-time network operators. The majority of the students who completed the assignment and survey had more than 5 years of experience in networking, and many had more than 15 years of experience. More than 200 of the students had experience with networks of more than 1,000 devices, and more than 300 of the students had experience with networks with more than 10,000 users. Most of the participants had the most experience with campus or enterprise networks.

The responses we received demonstrate a clear need for better tools for automating and verifying network control. Nearly 20% of participants said that they must change their network configurations more than once a day. The most common causes of changes were provisioning, planned maintenance, and updates to security policies—exactly the types of configuration changes that we aim to automate with Kinetic. More strikingly, 89% of respondents indicated that they were never completely certain that their changes to the configuration would not introduce a new problem or bug, and 82% were concerned that the changes would introduce problems with *existing* functionality that was unrelated to the change. The two most common aspects of configuration that operators wanted to see *automated* were correctness testing (37%) and quality of service and performance assurances (24%). The two most common aspects of configuration that participants wanted to see *verified* were general correctness problems (37%) and security properties (26%). We asked these same participants to write programs in Kinetic and other SDN controllers; we discuss the results of that part of our user study in Section 5.1.

2.2 Background: Pyretic and CTL

Pyretic. To develop a language for expressing control dynamics that is both concise and easy to use, we based the Kinetic language on Pyretic [25], a Python-embedded domain-specific programming language for writing SDN control programs. It encodes network data-plane behavior in terms of policy functions that map an incoming "located packet" (*i.e.*, a packet and its location) to an outgoing set of located packets. Pyretic has a `policy` variable that determines the actions that the control program applies to incoming packets (*e.g.*, filtering, modification, forwarding). Pyretic ultimately compiles policies to OpenFlow-based switches. Pyretic's composition operators provide straightforward mechanisms for composing multiple distinct policies into a single coherent control program. Pyretic's parallel composition operator (+) makes a copy of the original packet and applies the corresponding policies to each copy in parallel. Sequential composition (>>) applies policies to a packet in sequence, so that the second policy is applied to the packet that is the output of the first policy. Pyretic is extensible, and its support for composing distinct policies and dynamically recompiling flow-table entries whenever the `policy` variable is updated are useful features for Kinetic. Still, the language itself does not provide a framework for writing concise, intuitive policies that respond to changing conditions, which is Kinetic's goal.

Operator	**Meaning**
	(*Quantifiers over Groups of Paths*)
A ϕ	ϕ holds for all possible paths from the current state.
E ϕ	There exists a paths from the current state where ϕ holds.
	(*Quantifiers over a Specific Path*)
X ϕ	ϕ holds for ne**X**t state.
F ϕ	ϕ eventually holds sometime in the **F**uture.
G ϕ	ϕ holds for all current and following states, **G**lobally.
ϕ U ψ	ϕ holds at least **U**ntil ψ.

Table 2: *Computation tree logic (CTL) operators.*

Model checking. We wanted to design Kinetic so that policies were not only easy to automate, but also easy to verify. To do so, we applied a model checking framework developed by Clarke and Emerson [5, 6] and subsequently refined by McMillan [24]. Model checking can guarantee that a finite state machine (FSM) satisfies certain properties that are expressed in different types of logics; this feature makes FSMs a logical choice for expressing Kinetic policies. One such logic is computation tree logic (CTL), a branching-time logic that represents time as a tree structure. The initial state of an FSM is the root, and each node represents a different future state. A path through the tree represents an execution path of the FSM. CTL allows the expression of various types of temporal logic statements, such as those expressed in Table 2. NuSMV is a widely used symbolic model checker for FSMs [4]. The Kinetic compiler automatically translates Kinetic programs into an SMV model, which can be tested against various CTL-based assertions.

3 Kinetic by Example

We illustrate various features of Kinetic by way of example programs. All of the examples that we present in this section are verifiable; we defer a discussion of verification, as well as the details of the Kinetic language and runtime, to Section 4. We have selected examples that demonstrate the design features of Kinetic; the Kinetic Github repository has more examples [21].

Kinetic programs capture control dynamics with a finite state machine (FSM) abstraction. To illustrate this abstraction, we start with a simple example involving intrusion detection. Although FSMs are intuitive, representing all possible network states in a monolithic FSM would result in state explosion; the second and third examples illustrate two abstractions that address this challenge: Located Packet Equivalence Classes (LPECs) and FSM composi-

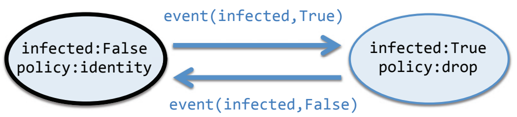


Figure 1: *Intrusion detection FSM.*

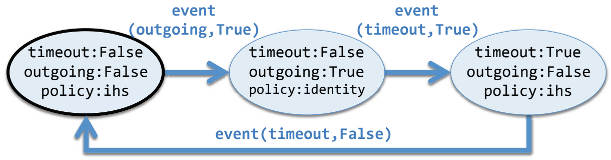


Figure 2: *Stateful firewall FSM.*

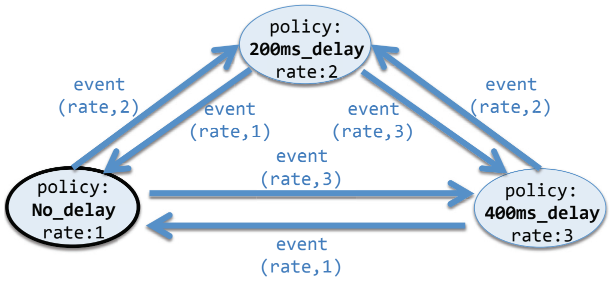


Figure 3: *Data usage-based rate limiter FSM.*

tion. Finally, to show Kinetic's generality, we present a MAC learning switch implementation.

3.1 Capturing Dynamics

We begin with a simple dynamic policy involving intrusion detection. Suppose that a network operator wants the network to drop all packets to and from a host once it receives an event indicating that the host is infected (*e.g.*, from an intrusion detection system). Kinetic allows operators to concisely express these dynamics with finite state machines that determine how a policy should evolve in response to events such as intrusions. We chose FSMs as the basic abstraction for expressing Kinetic programs because (1) they intuitively and concisely capture control dynamics in response to network events; and (2) their structure makes them amenable to verification.

In this example, each host would have a single state variable, `infected`. When `infected` is false, the controller applies Pyretic's `identity` (allow) policy for traffic from that host; when it is true, the controller applies Pyretic's `drop` policy for the host's traffic. Figure 1 shows this logical FSM. To support verification, the actual specification of the FSM for this policy is slightly more complicated; we expand on this example in Section 4.2.

3.2 Capturing State for Groups of Packets

Defining FSMs in Kinetic has the potential to create state explosion, since dynamic policies must be defined over a state space that is exponential in the number of hosts and flows (and possibly other aspects of the network). For example, consider the previous example, a two-state FSM indicating whether a host is infected. If the network has N hosts, then representing the state of the network requires an FSM with 2^N states, which is intractable, particularly as the size of the network and the complexity of policies grow. Instead of directly encoding an FSM that explicitly encodes all variable values, Kinetic encodes a single generic FSM that can be applied to any given group of packets (*e.g.*, all packets from the same host, in the case of the previous example). Each group of packets has a separate FSM instance; packets in the same group will always be in the same state. We call such a group of packets a *located packet equivalence class (LPEC)*.

To illustrate the use of LPECs, we describe the implementation of a stateful firewall that implements a common security policy. Figure 2 shows the Kinetic representation of the policy. This program always allows outbound traffic, but blocks inbound traffic unless the traffic flow is in response to corresponding outbound traffic for that flow. For example, if internal host ih_1 pings external host eh_2 then packets sent from eh_2 should be allowed back through the firewall until a certain timeout occurs, but only if ih_1 is the destination.

The firewall's initial state, in the left of the figure, shows the policy, `ihs`, which is a filter policy matching all traffic whose source address in the set of internal hosts. A Pyretic-encoded query collects outbound packets from hosts in `ihs` and produces (`outgoing,True`) event. This triggers the update of the `policy` variable to `identity` (indicating that traffic is now allowed), and `outgoing` is reset. The `timeout` event is provided by Kinetic event driver. After certain amount of time (*e.g.*, five seconds), a (`timeout,True`) event is invoked unless another outgoing packet is seen within the timeout. The program should regard inbound and outbound flows between the same pairs of endpoints with the same state, and the programmer should not have to explicitly encode state for every pair of endpoints. To implement such a policy, the programmer can define an LPEC to correspond to a distinct source-destination IP address pair:

```
def lpec(pkt):
  h1 = pkt['srcip']
  h2 = pkt['dstip']
  return (match(srcip=h1,dstip=h2) |
          match(srcip=h2,dstip=h1) )
```

3.3 Composing Independent Policies

Many aspects of network state are logically independent. For example, whether a host has authenticated is independent of whether it is infected or whether it has exceeded

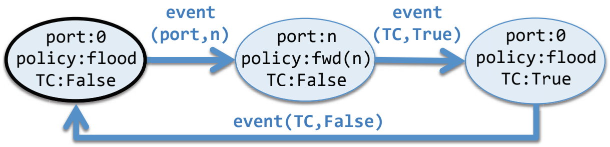


Figure 4: *MAC learner FSM.*

a usage cap. This independence allows a programmer to represent the overall network state as a product automaton that can be decomposed in terms of simpler *tasks*, where each task has simpler (and smaller) FSMs. This example shows the composition of four independent network tasks.

In our survey of campus network policies, we found nearly 20 university campuses [22] that implemented some form of usage-based rate-limiting (*e.g.*, [7]). Network operators currently implement these policies using low-level scripts that interact with monitoring devices. Kinetic provides intuitive mechanisms for implementing such a policy. Figure 3 illustrates the FSM for a usage-based rate limiter, which forward traffic with different delays depending on the user's historical data usage patterns. By default, traffic is forwarded with no delay; depending on the events that the controller receives concerning usage, the controller may institute a policy that introduces additional delay on user traffic. (OpenFlow 1.0 does not support traffic shaping, so we use variable delay as an illustrative example; Kinetic could be coupled with controllers that support later versions of OpenFlow that can do traffic shaping.)

Naturally, a real network would not only have policies involving quality-of-service, but also other policies, such as those relating to authentication and security. For example, a control program might first check whether a host is authenticated, either through a Web login or via 802.1X mechanism. Subsequently, the host's traffic might be subject to an intrusion detection policy that allows traffic by default but blocks the traffic if an infection event occurs. Finally, it might be sequentially composed with the rate-limiting policy above, yielding the resulting policy:

```
(web_auth + 802.1X_auth) >> ids >>
    rate_limiter
```

To verify this program, Kinetic generates a single FSM model for input to a model checker. Thus, programmers can write CTL specifications for the resulting composed policy, not only for individual policies. For example, a logic statement involving the combination of policies such as "If a host is authenticated either by the web authentication system or with 802.1X and is not infected, the resulting policy should never drop packets" can be verified with a single CTL assertion, as shown in Table 3. (Section 4.4 discusses verification in more detail.)

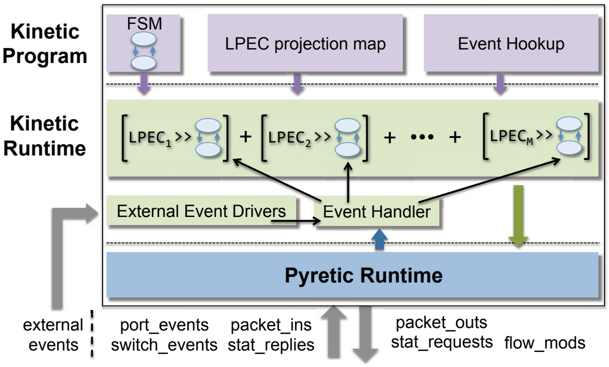


Figure 5: *Kinetic architecture.*

3.4 Handling General Event Types

Figure 4 shows a Kinetic FSM for a MAC learner that responds to both packets from hosts and topology changes. Although the implementation of a MAC learning switch is just as simple in other languages (indeed, it is the "canonical" reference program for SDN controllers), we present this example to illustrate that Kinetic programs can handle a variety of event types, including packet arrivals.

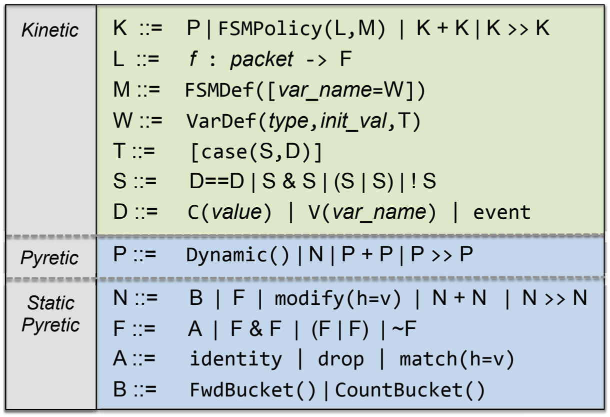
This program responds to two different types of events: `TC` (topo_change) and `port` events. The `TC` event is a built-in event that is invoked automatically whenever a topology change occurs. In Kinetic, programs can register and react to this built-in event. The `port` events are generated by a Pyretic query that collects the first packet for each (switch,srcmac) pair. The values of `policy` are defined by that of `port`: the value is `flood` when `port` is 0, and `fwd(n)` when `port=n`. Initially `port` is 0 (indicating the port has not yet been learned), and `TC` is `False`. When a `(port,n)` event arrives, which is invoked by the Pyretic runtime when it sees a packet from an unseen host, a transition occurs, setting the port to the value learned and the policy to unicast out that port. The MAC learner then unicasts packets to the appropriate hosts until a topology change occurs, triggering the transition to the right-most state in which `TC` is `True`, resulting in flooding for packets corresponding to that LPEC (*i.e.*, switch-source MAC address pair).

4 Kinetic Design & Implementation

We describe the details of Kinetic's architecture, language, runtime, and verification engine.

4.1 Architecture

We now describe the Kinetic system architecture, including the design of the Kinetic programming language. Figure 5 shows the Kinetic architecture, which is built on the Pyretic runtime. At the highest level, a Kinetic program



Kinetic	K ::= P | FSMPolicy(L,M) | K + K | K >> K L ::= *f* : *packet* -> F M ::= FSMDef([*var_name*=W]) W ::= VarDef(*type*,*init_val*,T) T ::= [case(S,D)] S ::= D==D | S & S | (S | S) | ! S D ::= C(*value*) | V(*var_name*) | event
Pyretic	P ::= Dynamic() | N | P + P | P >> P
Static Pyretic	N ::= B | F | modify(h=v) | N + N | N >> N F ::= A | F & F | (F | F) | ~F A ::= identity | drop | match(h=v) B ::= FwdBucket() | CountBucket()

Figure 6: *The Kinetic language grammar.*

has three parts: (1) a finite state machine (FSM) specification; (2) a specification of portions of flow space that are always in the same state in any given FSM (an LPEC); and (3) mechanisms for incorporating external events that could change the state of any given LPEC's FSM.

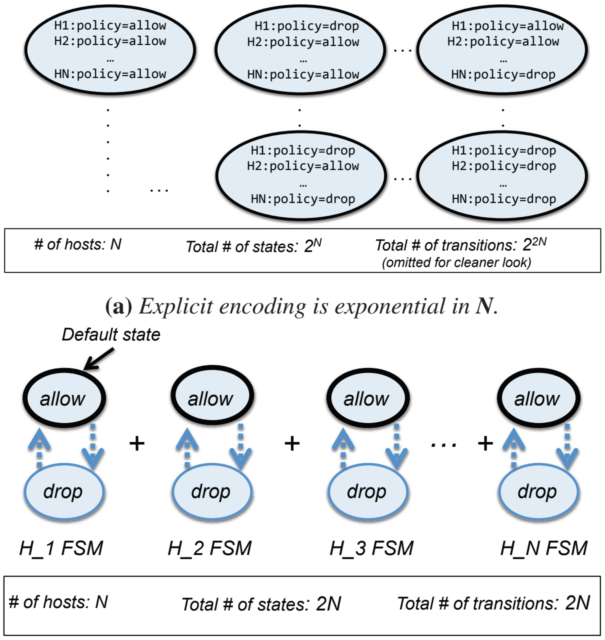
Kinetic instantiates copies of programmer-specified FSMs (one per LPEC); the Kinetic event handler sends incoming events, which can arrive either from external event hookups or from the Pyretic runtime (*e.g.*, in the case of certain types of events such as incoming packets), to the appropriate FSMs. Kinetic FSMs register with one or more event drivers and update their states when new events arrive, responding to incoming events that may be processed by those drivers. Kinetic supports both native events and generic JSON events. Because Kinetic is embedded in Pyretic, these functions can be executed using Pyretic's runtime. We use the Pyretic runtime to exchange OpenFlow messages with the network switches; we also use the Pyretic runtime to handle certain types of events, such as those related to either network topology or traffic.

4.2 Language and Abstractions

We offer a complete description of the language and then discuss LPECs and FSM composition in more detail.

4.2.1 Language Overview

Figure 6 defines the Kinetic language, which extends Pyretic (**P**). Pyretic has *bucket* policies (notated by **B**) which collect packets and count packet statistics, respectively; *primitive filters* (**A**) and *derived filters* (**F**) that allow only matching packets through; and *static policies* (**N**). Static policies include buckets, filters, the modify policy, and the combination of these via parallel and sequential composition. Dynamic generates a stream of static policies and can be combined with other policies in parallel or sequence.



(a) *Explicit encoding is exponential in N.*

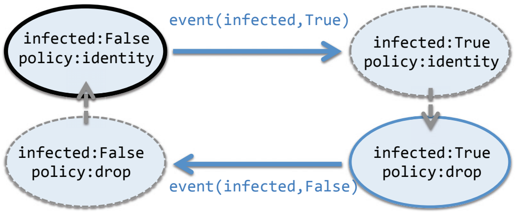
(b) *Decomposing to N LPEC FSMs.*

Figure 7: *Reducing state explosion using an LPEC FSM.*

Kinetic extends the Pyretic DSL with a subclass of Dynamic—FSMPolicy—which takes two arguments: an LPEC projection map (**L**) and an FSM description (**M**). The LPEC projection map takes a packet and returns a filter policy. The FSM description is set of assignments from a variable name to a variable definition (**W**). Each variable is defined by its type, initial value, and associated transition function (**T**). Each transition function is a list of cases, each of which contains a test (**S**) and an associated basic value (**D**) to which this corresponding state variable will be set, should this case be the first one in which the test is true. Tests are the logical combination of other tests (using *and*, *or*, *not*) or equality comparison between basic values. Finally, basic values are constants (**C(value)**), state variables (**V(variable_name)**), and events (**event**).

4.2.2 Located Packet Equivalence Classes

Recall from Section 3 that an LPEC allows an operator to encode a generic FSM for groups of packets (*e.g.*, all packets with the same source MAC address). Each distinct LPEC will have its own FSM instance, and the group of packets in each LPEC will be in the same state. Because each LPEC refers disjoint sets of packets, their FSMs (and corresponding policies) can be maintained independently, thus allowing their policies to be encoded in parallel. This mechanism allows the programmer to avoid explicit encoding of all combinations of network states (as shown in Figure 7a) and instead express each LPEC's FSM in-

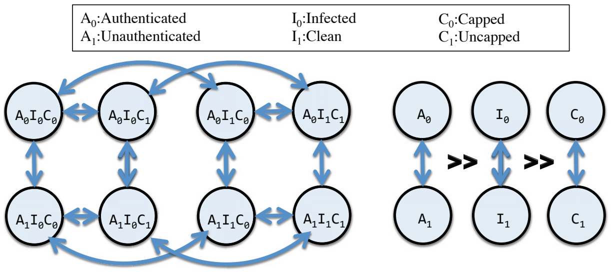


(a) *Actual implementation of the Kinetic FSM.*

```
1   @transition
2   def infected(self):
3     self.case(occurred(self.event),self.event)
4
5   @transition
6   def policy(self):
7     self.case(is_true(V('infected')),C(drop))
8     self.default(C(identity))
9
10  self.fsm_def = FSMDef(
11    infected=FSMVar(type=BoolType(),
12                    init=False,
13                    trans=infected),
14    policy=FSMVar(type=Type(Policy,{drop,identity}),
15                  init=identity,
16                  trans=policy))
17
18  def lpec(pkt):
19    return match(srcip=pkt['srcip'])
20
21  fsm_pol = FSMPolicy(lpec,self.fsm_def)
```

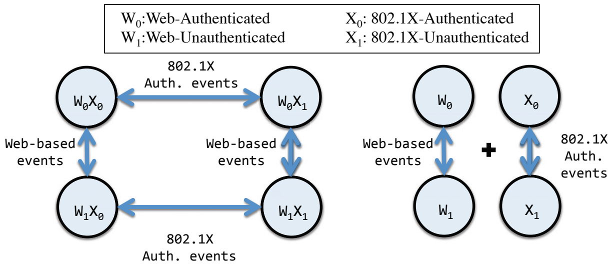
(b) *Kinetic code that implements the Kinetic FSM.*

Figure 8: *Logical FSM for an IDS in Kinetic, and the Kinetic code that implements the policy.*



(a) Without composition *(b) With composition*

Figure 9: *Composing independent tasks in sequence.*



(a) Without composition *(b) With composition*

Figure 10: *Composing multiple authentication tasks in parallel. Any successful authentication would result in allowing the host's traffic.*

dependently and compose them in parallel, as shown in Figure 7b.

Each LPEC has an FSM, which has a set of states, where each state has a Pyretic policy; and a set of transitions between those states, where transitions occur in response to events that the operators defines. When events arrive, the respective LPEC FSMs may transition between states, ultimately inducing the Pyretic runtime to recompile the policy and push updated rules to the switches. In Kinetic, a programmer can specify an LPEC in terms of a Pyretic filter policy. For example, `match(srcip=pkt['srcip'])` defines an LPEC FSM for each unique source IP address.

Returning to our IDS example from Section 3 (Figure 1), Figure 8b shows the code for the Kinetic program that implements the simple intrusion detection example from Section 3. Each host (*i.e.*, source IP address) can have a distinct state, so we need an LPEC FSM per source IP address; lines 18–19 define the LPEC. To define an FSM that is amenable to model checking, we must separate the `infected` variable and the corresponding `policy` variable into two separate states, as shown in Figure 8a. *Exogenous* events trigger transitions between the `infected` variable states; a change in this variable's value in turn triggers an *endogenous* transition of the `policy` variable, which ultimately causes the Pyretic runtime to recompile flow-table entries for the network switches. Lines 1–3 in Figure 8b define the exogenous transition for `infected`; lines 5–8 defined the endogenous transition for `policy` (note that the value of `policy` is defined in terms of the value of `infected`). Finally, lines 10–16 define the FSM itself, in terms of the two variables; the FSM definition is simply a set of FSM variables, each of which has a type, an initial value, and a transition function.

4.2.3 FSM Composition

In Section 3, we showed an example of a campus network policy that composed FSMs for independent network tasks to control state explosion. Without FSM composition, a programmer would need to define FSMs for $\Pi_{i=1}^{N} a_i$ possible states, where a_i is the number of possible states for task i and N is the total number of tasks. Decomposing the product automaton reduces state complexity from exponential to linear in the number of independent tasks. For example, given ten tasks, each with two states, a monolithic program would require 1,024 states, as opposed to just 20.

Pyretic allows policies to be composed either in parallel (*i.e.*, on independent copies of the same packet) or in sequence (*i.e.*, where the second policy is applied to the output from the first). It turns out that these operators are *also* useful for reducing state explosion. Figure 9 illustrates how sequential composition can reduce state complexity by decomposing a larger product automaton. Consider a simple control program that puts the host into a walled-garden until it has authenticated, quarantines the host if an infection has been detected, and rate limits a host if it has exceeded a usage cap. Each of these tasks has two possible states: authenticated or not, quarantined or not, capped or not, resulting in 2^3 possible states. By applying `auth >> IDS >> cap`, the same network control program requires only $2 \cdot 3$ states.

Figure 10 shows how parallel composition reduces state complexity. A Kinetic program might specify that either a network flow should be authenticated by a Web authentication mechanism or 802.1X. If *either* of these tasks places the host in an authenticated state, the host should be allowed to send traffic. Without composition, the network state machine would need a second set of states, requiring 2^N states, where N is the number of authentication tasks (in this case, $N = 2$). (Clearly, even more states would be needed if any independent task could assume more than two states.) As before, decomposition reduces this to $\Sigma_{i=1}^{N} a_i$ states, where a_i is the number of possible states for task i, and N is the number of tasks.

4.3 Runtime

We now explain optimizations to the Kinetic runtime to support the efficient compilation of the large finite state machines that might result from networks with many hosts and policies and high event rates. The Kinetic runtime's main challenge is storing and processing the joint state of all LPEC FSMs to produce a single set of forwarding table entries in the network switch. To accomplish this goal, the runtime first decomposes the FSMs with combinators to achieve a representation of the network state that is linear in the number of hosts and policies. Second, Kinetic optimizes the compilation process itself by recognizing that the LPEC FSMs typically operate on disjoint flow space, which allows for optimizations that dramatically speed up parallel composition. Finally, Kinetic only expands the LPEC FSMs for which a transition has actually occurred. We describe each of these optimizations below.

Decomposing the product automata. A Kinetic `FSMPolicy` encodes the complete FSM as the *product automaton* [10] of the individual LPEC FSMs. We can represent the Pyretic policy for the entire network, given a global network state s as the following product automata:

$$\text{policy} \quad = \quad \sum_{i=1}^{N} (\text{lpec}_\text{i} \ >> \ \text{lfsm}_i(s))$$

where the summation operator represents parallel composition of the corresponding policies, and each LPEC generator produces the appropriate packets that are processed by the corresponding LPEC FSM in state s.

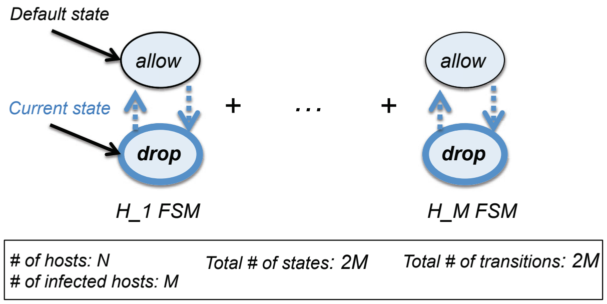


Figure 11: *Expanding only **M** LPEC FSMs that have changed.*

Fast compilation of disjoint LPECs. Compilation of policies that are composed in parallel is computationally expensive, as it requires producing the cross-product of all match and action rules: it involves computing the intersection of match statements and the union of actions, for every pair of match and action pairs between the two policies. If LPECs are disjoint, however, the resulting policies can simply be combined without explicitly computing the intersection of the match statements: the rules from each LPEC FSM can simply be inserted into the flow tables.

Default policies and on-demand LPEC FSM expansion. Even a linear-sized representation may not scale. For example, the LPEC generator shown in Section 4.2.2 would generate 2^{32} LPECs if it were fully expanded, while a generator for each pair of hosts (based on hardware address) would produce 2^{96} LPECs. Fortunately, because all LPEC FSMs are generated from the same FSM specification, they start with the same initial state and, hence, the same default policy. Thus, Kinetic does not need to expand the FSM for an LPEC *unless and until it experiences a state transition*; until that point, the Kinetic runtime can simply apply whatever default policy is defined for that FSM. Figure 11 highlights this on-demand expansion.

Kinetic's runtime optimizations reduce the computational complexity of compilation from exponential in the number of LPECs (Figure 7a), to linear in the number of LPECs (Figure 7b), and finally to linear in the (much smaller) number of LPECs that have actually experienced a transition (Figure 11). Kinetic additionally employs additional optimizations, such as memoizing previously compiled policies, as other applications have used [15].

4.4 Verification

When the programmer executes a Kinetic program, Kinetic automatically creates an FSM model for the NuSMV model checker. Kinetic obtains information about each state variable (*e.g.*, type, initial value, and transition relationship) by parsing the `fsm_def` data structure; Kinetic

```
1  MODULE main
2    VAR
3      policy    : {identity,drop};
4      infected  : boolean;
5    ASSIGN
6      init(policy)    := identity;
7      init(infected)  := FALSE;
8      next(infected)  :=
9        case
10         TRUE      : {FALSE,TRUE};
11       esac;
12     next(policy) :=
13       case
14         infected  : drop;
15         TRUE      : identity;
16       esac;
```

Figure 12: *NuSMV FSM model for IDS policy from Figure 8b.*

parses the transition function for additional information about transitions, which often depend on other variables.

Kinetic then uses NuSMV to test CTL specifications that the programmer writes against the FSM model. Kinetic outputs the CTL specifications that passed; for any failed specifications, Kinetic produces a counterexample, showing the sequence of events and variable changes that violated the specification. In addition to single `FSMPolicy` objects, Kinetic can convert composed policies into a single model that can be verified. For example, although the programmer specifies a composed policy as in Figure 9b and Figure 10b, verification will execute on a combined FSM model as in Figure 9a and Figure 10a.

Figure 12 shows the NuSMV FSM model corresponding to the IDS policy from Figure 8b. The model definition has two parts. The first is `VAR`, which declares the names and types of each variable (lines 2–4). The second is `ASSIGN`, where current and future variable values are assigned, using two functions for each variable: an `init` function that determines the variable's initial value (line 5–7), and a `next` function that specifies what value or values the variable may take, as a function of the current values of other variables in the model (line 8–16).

Within the `case` clause of each `next` function, the left-hand side shows the condition, while the right-hand side shows the variable's next value if the condition holds. `TRUE` on the left-hand side refers to a default transition. Lines 8–11 indicate that the `infected` variable can change between `FALSE` and `TRUE`, independent of any other state variable (in reality, the value changes based on external event of the same name). The `policy` variable in lines 12–16 shows that the value transitions to `drop` if `infected` is `True`, while the default is `identity`. Thus, it shows that `policy`'s next value depends on the `infected` variable. Table 3 shows examples of the types of temporal properties that Kinetic can verify.

5 Evaluation

In this section, we evaluate two aspects of Kinetic: (1) Does Kinetic make it easier for network operators to configure realistic network policies? (Section 5.1); and (2) How does Kinetic's performance scale with the number of flows, users, and policies? (Section 5.2).

5.1 Programming in Kinetic: User Study

Evaluating whether a new network configuration paradigm such as Kinetic makes it easier for network operators to write network policies is challenging. Network operators already know how to use existing tools and infrastructure, and deploying a new control framework requires overcoming both the inertia of network infrastructure that is already deployed *and* the knowledge base of network operators, many of whom are not programmers by training. We needed to find a way to ask many network operators to evaluate Kinetic in light of these obstacles. Fortunately, the Coursera course on software-defined networking that we teach [8] offers precisely this captive audience, as the course's demographic includes many network operators who are both educated about SDN and willing to experiment with cutting-edge tools. (Section 2 and Table 1 explained the initial survey and described the demographics of the participants.) We obtained approval from our institutional review board (IRB) to ask students to use Kinetic and other SDN controllers to complete a simple network management task and subsequently survey them.

We asked the students in the course to write a "walled garden" controller program that is inspired from real enterprise network management task that we have learned about in our discussions with network operators [9]. In summary, the students were asked to write a program that permitted all traffic to and from the Internet unless a host was deemed to be infected (*e.g.*, as determined from an intrusion detection system alert) *and* not a host that was exempt from the policy (one might imagine that certain classes of users, such as high-ranking administrators or executives would get different treatment than strict interruption of service). In the assignment, we asked the students to: (1) Write a Kinetic program that implements the policy; (2) Choose either Pyretic or POX to implement the same policy; (3) Optionally implement the policy in the remaining controller; (4) Answer survey questions about their experiences with each controller.

The course devoted one week each to each of the three controller platforms, and students had already completed assignments in both POX and Pyretic, so if anything, students should have found those platforms at least as familiar as Kinetic. In fact, there were three programming assignments in Pyretic while there was only one for Kinetic. Kinetic was discussed in only one lecture out of eight

Program	CTL	Description
Mac learner	AG (topo_change → AX policy=flood)	Always resets to flooding when topology changes.
	AG (policy=flood → AG EF (port>0))	Can always go from flooding to unicasting a learned port.
	! AG (port=1 → EX port=2)	It is impossible to update the learned port without first flooding.
	AG (port>0 → A [port>0 U topo_change])	Port will stay learned until there is a topology change.
Stateful firewall	AG (outgoing & !timeout → AX policy=identity)	If first packet originated from internal host and timeout did not occur, the system should allow traffic.
	AG (outgoing & timeout → AX policy=matchFilter)	If first packet originated from internal host, but timeout occurred, the system should shut down traffic (apply match filter).
	AG (!outgoing → AX policy=matchFilter)	If first packet is not from internal host, the system should not allow traffic (apply match filter).
Composed policy	AG (infected → AX policy=drop)	If host is infected, drop packets.
	AG ((authenticated_web | authenticated_1x) & !infected → AX policy!=drop)	If host is authenticated either by Web or 802.1X, and is not infected, packets should never be dropped.
	AG (authenticated_web & !infected & rate=2 → AX policy=delay200)	If host is authenticated by Web, not infected, and the rate is 2, delay packets by 200ms.

Table 3: *NuSMV CTL rules for different Kinetic programs.*

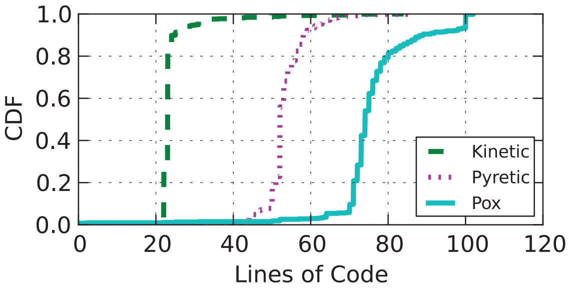


Figure 13: *The lines of code required to implement the walled-garden program in different controller languages.*

Programs	FL	Pox	Pyretic	Kinetic
ids/firewall	416	22	46	17
mac_learner	314	73	17	33
server load balancer	951	145	34	37
stateful firewall	–	–	25	41
usage-based rate limiter	–	–	–	30

Table 4: *Lines of code to implement programs in each controller.*

lectures in the course, and was not treated specially. To further minimize the bias in favor of Kinetic, students were instructed to complete the assignment in Kinetic first, as the first attempt is usually the hardest. With better understanding of the assignment, it is likely that programming in Pyretic or POX would have been easier.

Of the students who completed the survey from Section 2, 667 attempted the assignment, and 631 successfully completed it (a 95% completion rate), and 70% of those students completed the programming assignment in less than three hours. We asked students who did not complete the assignment why they did not complete it; most referenced external factors such as time constraints, as opposed to anything pertinent to Kinetic.

To compare the complexity of the different control programs, we compare the lines of code in programs implemented with different controllers; we then conduct qualitative measurements by surveying the students of the course. Although the lines of code for a program depends on the language, programmer, and implementation style, a high-level comparison can nevertheless yield a rough but meaningful sense for the relative simplicity of a Kinetic program. Figure 13 shows the distribution of the lines of code that students needed to implement the walled-garden program in different controller languages. About 80% of the implementations using Kinetic required about 22 lines of code; in contrast, more than half of the Pyretic implementations required more than 50 lines, and half of the assignments written in POX required more than 75 lines of code. The fact that Kinetic requires fewer lines of code to implement this program highlights the utility of the abstractions that Pyretic provides. In addition to the experiment from the Coursera course, where we could find publicly available implementations on the Web, we compared the number of lines of code for several different programs to our own implementations of the same programs in Kinetic. (Blank entries in the table indicate that no implementation was available.) Table 4 shows these results, for four different controllers: Floodlight, POX, Pyretic, and Kinetic. The public Pyretic programs are occasionally slightly shorter than the corresponding Kinetic programs because they only handle built-in events such as packet arrivals and topology changes. Programs that need to handle arbitrary events would likely always be shorter in Kinetic.

In addition to analyzing quantitative measures such as lines of code, we asked students more qualitative questions about their experiences using Kinetic to implement the walled-garden assignment, relative to their experiences with Pyretic and POX. We asked students to rank the controllers based on ease of use, as well as which platform they preferred. Figure 14 shows some highlights from this part of the survey. Of the three controllers, more than half of the students preferred writing the assignment in Kinetic

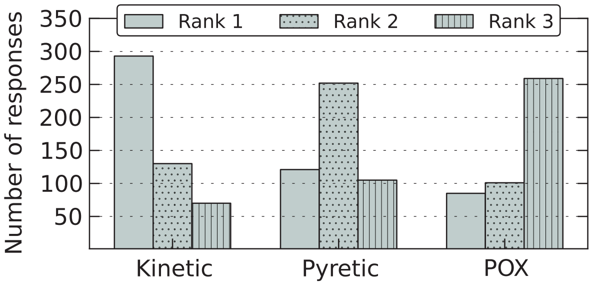


Figure 14: *Number of students who preferred each controller.*

versus either Pyretic or POX. We asked students whether Kinetic could make it easier to configure and verify policies in their networks. About 79% of students thought that configuring the network with Kinetic would be easier than current approaches, and about 84% agreed that Kinetic would make it easier to verify network policies.

Students who chose Kinetic as the language that they preferred best cited its abstractions, FSM-based structure, and support for intuition (*e.g.*, "Kinetic is more intuitive: the only thing I need to do is to define the FSM variable.", "intuitive and easy to understand", "reduces the number of lines of code", "programming state transitions in FSMs makes much more sense", "the logic is more concise"). Some students still preferred Kinetic, despite the fact that the syntax had a steeper learning curve: "Kinetic took less time and was actually more understandable using the templates even though the structure was very 'cryptic'... I thought the Pyretic would be the easiest...[but] I spent a lot more time chasing down weird bugs I had because of things I left out or perhaps didn't understand." Interestingly, many of the students actually preferred the lower-level trappings of POX to Pyretic (*e.g.*, "Pyretic was friendly, but the logic more intricate"). The results of this experiment and survey highlight both the advantages and disadvantages of Kinetic's design, as well as the difficulty of designing "northbound" languages for SDN controllers: without intuitive abstractions, operators may even *prefer* the lower-level APIs to higher-level abstractions.

5.2 Performance and Scalability

We evaluate Kinetic's performance and scalability when handling incoming events, as well as the performance and scalability of verification, as those are the two main contributions of our work. We evaluated the Kinetic controller on a machine with an Intel Xeon CPU E5-1620 3.60 GHz processor and 32 GB of memory. We measured raw packet forwarding performance but do not focus those numbers, as the forwarding performance is not the focus of our work and is equivalent to what can be achieved with POX and Pyretic, in any case. Similarly, the rate at which updated rules can be installed depends on lower layers (*e.g.*, POX). Optimizing the number of rule updates [32] and applying them consistently [28] have been studied in previous work, so we do not focus on those aspects here.

Statistic	# per day
Total Unique Authenticated Users	22,586
Total Unique Devices Authenticated	41,937
Number of WPA authentication Events	1,330,220
Number of WEB authentication Events	1,850

Table 5: *The frequency of network events on a primary campus network, which we use for a trace-driven evaluation of Kinetic.*

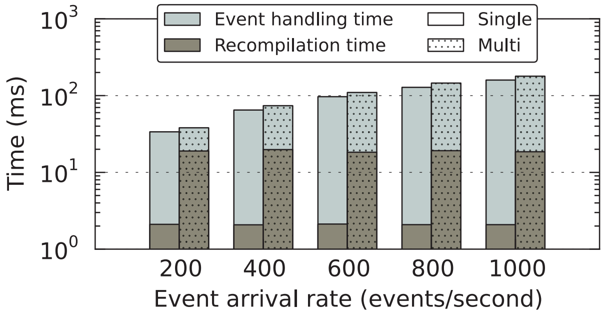


Figure 15: *Time to handle a batch of incoming events and recompile policies in Kinetic, for different event arrival rates and policies.*

Event handling and policy recompilation. Because Kinetic recompiles the policy whenever an event causes a state transition, we must evaluate how fast Kinetic can react to events and recompile the policy for realistic network scenarios. We used the wireless network from a large university campus with more than 4,300 access points deployed across 200 buildings; the network authenticates nearly 42,000 unique devices for more than 22,000 users every day. Table 5 summarizes these statistics. On such a network, Kinetic would have to keep track of an equivalent number of devices, and each authentication event (about 1.3 million per day) would require Kinetic to recompile policy, resulting in an average of 15 events per second (though certainly higher during peak periods). We evaluate Kinetic for event arrival rates for up to 1,000 events per second, for both a single-FSM policy and a policy involving the composition of multiple FSMs, based on the example in Section 3.3. We create a Kinetic program that results in 42,000 LPEC FSMs and randomly distribute authentication events across these FSMs (*i.e.*, devices).

Figure 15 presents the results of this experiment. Recompilation time is longer for the program with multiple FSMs composed together as it embeds a more complex policy than the program with a single FSM. For both programs, event handling time increases as event arrival rate increases. Even for event arrival rates that are several orders of magnitude more frequent than an actual campus

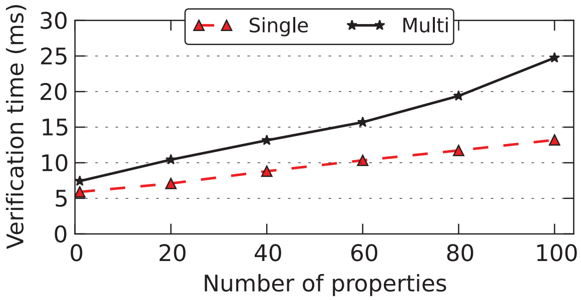


Figure 16: *Verification time as a function of the number of CTL properties that Kinetic checks, for a policy with a single FSM and a policy with a composition of multiple FSMs.*

network, Kinetic's event handling and recompilation times are around (or less than) a few hundred milliseconds.

Verification. The speed of verification depends on the performance of NuSMV, which in turn depends on three factors: (1) the size of the given FSM (*i.e.*, number of states and transitions), (2) the number of properties to verify, and (3) the kinds of properties to verify. To observe whether Kinetic's verification time is reasonable, we evaluate Kinetic's verification performance with the same programs that we used to evaluate Kinetic's event handling and recompilation performance. The single-FSM authentication program produces an FSM with four states, and the program with multiple FSMs produces a combined FSM with 384 states. To test Kinetic with more than the handful of CTL specifications we manually created, we generate over one hundred specifications using random combinations of CTL operators. They are all syntactically correct (*i.e.*, NuSMV will not complain about the syntax), but are generated regardless of whether they will be true or false when each goes through the model checker, as our goal is merely to measure verification time.

Figure 16 shows the time to complete verification for different Kinetic programs with different numbers of properties. Each experiment had 1,000 independent trials; the variance across experiments was small, so we do not show the error bars. As expected, a Kinetic program with a larger FSM model takes longer to finish. The figure also shows that the number of properties affects verification time, but all verification finishes within 35 milliseconds. Kinetic performs verification before the Kinetic program ever runs, so this process has no effect on performance.

6 Related Work

We discuss SDN controllers with verifiable properties, approaches for formally verifying data-plane behavior, and other high-level SDN control languages.

Formal verification of SDN control programs. FlowLog [26] provides a database-like programming model that unifies the control-plane logic with data-plane state and controller state. Aspects of FlowLog programs can be verified, but because the language does not naturally capture state transitions and temporal relationships, it cannot verify arbitrary temporal relationships, such as those that can be verified with CTL in Kinetic. FlowLog uses Alloy to perform bounded verification, so its analysis is not complete, and certain aspects of verification are manual. FlowLog has not been evaluated for realistic network policies or for large networks. It requires storing multiple database entries for each network state variable and handles certain aspects of control logic by sending data packets to the controller, so it is unlikely to scale. VeriCon [1] verifies that a program written in its language (CSDN) is correct for all topology and packet events (*e.g.*, packet arrivals, switch joins). It does not handle arbitrary network events, and there is no OpenFlow-based implementation, so its practicality is unclear.

Formal verification of data-plane behavior. Recent work in network verification has focused on verifying static properties of the data-plane state [19, 28, 29]. Anteater [23] and HSA [17, 18] can verify properties of a static snapshot of a network's data-plane state. These systems can determine whether a static snapshot of data-plane state violates some invariant, but they do not verify the logic of the control program that generated the state in the first place, making it difficult to identify which aspect of the network's control-plane logic caused the incorrect data-plane state. In contrast, Kinetic helps operators verify control logic, such as "if an intrusion detection system determines that a host is infected, the host's traffic should be dropped". This capability helps operators both reason about future data-plane states that a control program could install and troubleshoot incorrect behavior when it does arise. Because Kinetic's verifies the static programs themselves, it can detect logic errors before the control program is ever run on a live network. NICE [2] can test control-plane properties that might result from arbitrary sequences of standard OpenFlow events; it is not a controller, but rather a test harness for control programs written in existing low-level controllers (*e.g.*, NOX) and hence does not permit reasoning about arbitrary events.

Other SDN control languages. Many languages raise the level of abstraction for writing network control programs, yet these languages do not offer constructs for concisely encoding policies that capture network dynamics, nor do they incorporate formal verification of control-plane behavior. FML [16] allows network operators to write and maintain policies in a declarative style. Nettle [30] is a domain specific language in Haskell. Procera [31] applies functional reactive programming to help operators express policies. Frenetic [12] is a family of languages that share fundamental constructs and techniques for efficient compilation to OpenFlow switches.

7 Conclusion

One of the reasons that network configuration is so challenging is that network conditions are continually changing, and network operators must adapt the network configuration whenever these conditions change. Network operators need means not only to automate these configuration changes but also to verify that the changes will be correct. Existing general-purpose SDN controllers lack intuitive constructs for expressing dynamic policy and ways to efficiently verify that the control programs conform to expected behavior.

To address these problems, we designed and developed Kinetic, a domain specific language and SDN controller for implementing dynamic network policies in a concise, verifiable language. Kinetic exposes a language that allows operators to express network policy in an intuitive language that maps directly to a CTL-based model checker. We evaluated Kinetic's usability and performance through both a large-scale user study and trace-driven performance evaluation on realistic policies and found that network operators find Kinetic easy to use for expressing dynamic policies and that Kinetic can scale to a large number of policies, hosts, and network events.

Kinetic sits squarely in the realm of ongoing work on network verification and complements the growing body of work on data-plane verification, such as Veriflow [19] and NetPlumber [17]. As these tools can help network operators ask questions about snapshots of data-plane state, and Kinetic can help network operators reason about the dynamics of network policies (which ultimately compile to the corresponding data-plane state), the approaches are complementary. Similarly, Kinetic needs the path guarantees that consistent updates [28] provide to guarantee that the properties it verifies are preserved during state transitions; conversely, consistent updates could be extended to reason about temporal properties such as those that Kinetic can express. One natural next step would be to combine these approaches.

Acknowledgements

We thank the NSDI reviewers and our shepherd Dejan Kostic for providing valuable feedback for this paper. We also thank Jennifer Rexford, Nate Foster, David Walker, and Yoshio Turner for their helpful comments on earlier versions of this paper. This work was supported by NSF Awards CNS-1261357 and CNS-1409076.

References

[1] T. Ball, N. Bjørner, A. Gember, S. Itzhaky, A. Karbyshev, M. Sagiv, M. Schapira, and A. Valadarsky. Vericon: towards verifying controller programs in software-defined networks. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, page 31. ACM, 2014. (Cited on page 12.)

[2] M. Canini, D. Venzano, P. Peresini, D. Kostic, and J. Rexford. A NICE Way to Test OpenFlow Applications. In *USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, Apr. 2012. (Cited on page 12.)

[3] Chef. `http://www.opscode.com/chef/`. (Cited on page 1.)

[4] A. Cimatti, E. Clarke, E. Giunchiglia, F. Giunchiglia, M. Pistore, M. Roveri, R. Sebastiani, and A. Tacchella. NuSMV 2: An Open-source Tool for Symbolic Model Checking. In *Computer Aided Verification*, pages 359–364. Springer, 2002. (Cited on pages 2 and 3.)

[5] E. M. Clarke, E. A. Emerson, and A. P. Sistla. Automatic verification of finite-state concurrent systems using temporal logic specifications. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 8(2):244–263, 1986. (Cited on page 3.)

[6] E. M. Clarke, O. Grumberg, and D. A. Peled. *Model checking*. MIT press, 1999. (Cited on pages 2 and 3.)

[7] CMU Network Bandwidth Usage Guidelines. `http://www.cmu.edu/computing/network/connect/guidelines/bandwidth.html`. (Cited on page 5.)

[8] Coursera: SDN class. `https://www.coursera.org/course/sdn`, 2014. (Cited on pages 2 and 9.)

[9] Coursera: SDN Walled Garden Assignment. `http://goo.gl/JYMret`, 2014. (Cited on page 9.)

[10] S. Eilenberg. *Automata, languages, and machines*, volume 1. Access Online via Elsevier, 1974. (Cited on pages 2 and 8.)

[11] N. Feamster, J. Rexford, and E. Zegura. The Road to SDN. *Queue*, 11(12):20:20–20:40, Dec. 2013. (Cited on page 1.)

[12] N. Foster, M. J. Freedman, A. Guha, R. Harrison, N. P. Katta, C. Monsanto, J. Reich, M. Reitblatt, J. Rexford, C. Schlesinger, A. Story, and D. Walker. Languages for Software-Defined Networks. *IEEE Communications*, 51(2):128–134, Feb. 2013. (Cited on page 12.)

[13] N. Foster, R. Harrison, M. Freedman, C. Monsanto, J. Rexford, A. Story, and D. Walker. Frenetic: A network programming language. In *International Conference on Functional Programming (ICFP)*, Sept. 2011. (Cited on page 1.)

[14] Frenetic, Pyretic and Resonance. `http://blog.sflow.com/2013/08/frenetic-pyretic-and-resonance.html`. Note: Kinetic was previously known as Resonance. (Cited on page 2.)

[15] A. Gupta, L. Vanbever, M. Shahbaz, S. P. Donovan, B. Schlinker, N. Feamster, J. Rexford, S. Shenker, R. Clark, and E. Katz-Bassett. SDX: A Software Defined Internet Exchange. In *ACM SIGCOMM*, pages 579–580, Chicago, IL, 2014. (Cited on page 8.)

[16] T. L. Hinrichs, N. S. Gude, M. Casado, J. C. Mitchell, and S. Shenker. Practical declarative network management. In *ACM/USENIX Workshop on Research on Enterprise Networking (WREN)*, pages 1–10, Barcelona, Spain, 2009. (Cited on page 12.)

[17] P. Kazemian, M. Chang, H. Zeng, G. Varghese, N. McKeown, and S. Whyte. Real-time Network Policy Checking using Header Space Analysis. In *USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2013. (Cited on pages 12 and 13.)

[18] P. Kazemian, G. Varghese, and N. McKeown. Header Space Analysis: Static Checking for Networks. In *USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, Apr. 2012. (Cited on pages 2 and 12.)

[19] A. Khurshid, W. Zhou, M. Caesar, and P. B. Godfrey. Veriflow: Verifying network-wide invariants in real time. In *USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, Lombard, IL, Apr. 2013. (Cited on pages 2, 12 and 13.)

[20] H. Kim, T. Benson, A. Akella, and N. Feamster. The Evolution of Network Configuration: A Tale of Two Campuses. In *ACM SIGCOMM Internet Measurement Conference (IMC)*, pages 499–514, Berlin, Germany, 2011. (Cited on page 1.)

[21] Kinetic source code. `https://github.com/frenetic-lang/pyretic/tree/kinetic` Note: Evaluation specific data & scripts are in the kinetic_test branch. (Cited on pages 2 and 3.)

[22] List of Real Campus Network Acceptable Use Policies. `http://goo.gl/pdR6Sd`, 2014. (Cited on pages 1 and 5.)

[23] H. Mai, A. Khurshid, R. Agarwal, M. Caesar, P. B. Godfrey, and S. T. King. Debugging the data plane with anteater. In *ACM SIGCOMM*, pages 290–301, Toronto, Ontario, Canada, 2011. (Cited on page 12.)

[24] K. L. McMillan. *Symbolic model checking*. Springer, 1993. (Cited on page 3.)

[25] C. Monsanto, J. Reich, N. Foster, J. Rexford, and D. Walker. Composing Software-Defined Networks. In *USENIX NSDI*, 2013. (Cited on pages 2 and 3.)

[26] T. Nelson, A. D. Ferguson, M. J. Scheer, and S. Krishnamurthi. Tierless programming and reasoning for software-defined networks. In *USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, pages 519–531, Seattle, WA, Apr. 2014. (Cited on page 12.)

[27] Puppet. `http://puppetlabs.com/solutions/juniper-networks`. (Cited on page 1.)

[28] M. Reitblatt, N. Foster, J. Rexford, C. Schlesinger, and D. Walker. Abstractions for Network Update. In *ACM SIGCOMM*, pages 323–334, Helsinki, Finland, 2012. (Cited on pages 11, 12 and 13.)

[29] D. Sethi, S. Narayana, and S. Malik. Abstractions for model checking sdn controllers. In *Formal Methods in Computer-Aided Design (FMCAD), 2013*, pages 145–148, 2013. (Cited on page 12.)

[30] A. Voellmy and P. Hudak. Nettle: Taking the sting out of programming network routers. In *Practical Aspects of Declarative Languages (PADL)*, pages 235–249. Springer, 2011. (Cited on page 12.)

[31] A. Voellmy, H. Kim, and N. Feamster. Procera: a language for high-level reactive network control. In *ACM SIGCOMM Workshop on Hot Topics in Software Defined Networks (HotSDN)*, pages 43–48, Helsinki, Finland, 2012. (Cited on page 12.)

[32] X. Wen, L. Li, C. Diao, X. Zhao, and Y. Chen. Compiling Minimum Incremental Update for Modular SDN Languages. In *ACM SIGCOMM Workshop on Hot Topics in Software Defined Networking (HotSDN)*, pages 193–198. ACM, 2014. (Cited on page 11.)

Enforcing Customizable Consistency Properties in Software-Defined Networks

Wenxuan Zhou*, Dong Jin**, Jason Croft*, Matthew Caesar*, and P. Brighten Godfrey*

*University of Illinois at Urbana-Champaign
**Illinois Institute of Technology

Abstract

It is critical to ensure that network policy remains consistent during state transitions. However, existing techniques impose a high cost in update delay, and/or FIB space. We propose the Customizable Consistency Generator (*CCG*), a fast and generic framework to support *customizable* consistency policies during network updates. *CCG* effectively reduces the task of *synthesizing* an update plan under the constraint of a given consistency policy to a *verification* problem, by checking whether an update can safely be installed in the network at a particular time, and greedily processing network state transitions to heuristically minimize transition delay. We show a large class of consistency policies are guaranteed by this greedy heuristic alone; in addition, *CCG* makes judicious use of existing heavier-weight network update mechanisms to provide guarantees when necessary. As such, *CCG* nearly achieves the "best of both worlds": the efficiency of simply passing through updates in most cases, with the consistency guarantees of more heavyweight techniques. Mininet and physical testbed evaluations demonstrate *CCG*'s capability to achieve various types of consistency, such as path and bandwidth properties, with zero switch memory overhead and up to a 3× delay reduction compared to previous solutions.

1 Introduction

Network operators often establish a set of correctness conditions to ensure successful operation of the network, such as the preference of one path over another, the prevention of untrusted traffic from entering a secure zone, or loop and black hole avoidance. As networks become an increasingly crucial backbone for critical services, the ability to construct networks that obey correctness criteria is becoming even more important. Moreover, as modern networks are continually changing, it is critical for them to be correct even during transitions. Thus, a key challenge is to guarantee that properties are preserved during transitions from one correct configuration to a new correct configuration, which has been referred as *network consistency* [25].

Several recent proposed systems [13, 16, 19, 25] consistently update software-defined networks (SDNs), transitioning between two operator-specified network snapshots. However, those methods maintain only *specific* properties, and can substantially delay the network update process. Consistent updates [25] (CU), for example, only guarantees *coherence*: during a network update any packet or any flow is processed by either a new or an old configuration, but never by a mix of the two. This is a relatively strong policy that is sufficient to guarantee a large class of more specific policies (no loop, firewall traversal, etc.), but it comes at the cost of requiring a two-phase update mechanism that incurs substantial delay between the two phases and doubles flow entries temporarily. For networks that care only about a weaker consistency property, e.g., only loop freedom, this overhead is unnecessary. At the same time, networks sometimes need properties *beyond* what CU provides: CU only enforces properties on individual flows, but not across flows (e.g., "no more than two flows on a particular link"). SWAN [13] and zUpdate [19] also ensure only a specific property, in their case congestion freedom.

That leads to a question: is it possible to efficiently maintain *customizable* correctness policies as the network evolves? Ideally, we want the "best of both worlds": the efficiency of simply immediately installing updates without delay, but the safety of whatever correctness properties are relevant to the network at hand.

We are not the first to define this goal. Recently, Dionysus [15] proposed to reduce network update time to just what is necessary to satisfy a certain property. However, Dionysus requires a rule dependency graph for each particular invariant, produced by an algorithm specific to that invariant (the paper presents an algorithm for packet coherence). For example, a waypointing invariant would need a new algorithm. Furthermore, the algorithms work only when forwarding rules match exactly one flow.

We take a different approach that begins with an observation: synthesizing consistent updates for arbitrary consistency policies is hard, but network verification on general policies is comparatively easy, especially now that real-time data plane verification tools [5, 17, 18] can verify very generic data-plane properties of a network state within milliseconds. In fact, as also occurs in domains outside of networking, there is a connection between synthesis and verification. A feasible update sequence is one which the relevant properties are verifiable at each moment in time. Might a verifier serve as a guide through the search space of possible update sequences?

Based on that insight, we propose a new consistent update system, the Customizable Consistency Generator (*CCG*), which efficiently and consistently updates SDNs under customizable properties (invariants), intuitively by converting the scheduling synthesis problem to a series of network verification problems. With *CCG*, network programmers can express desired invariants using an interface (from [18]) which allows invariants to be defined as essentially arbitrary functions of a data plane snapshot, generally requiring only a few tens of lines of code to inspect the network model. Next, *CCG* runs a greedy algorithm: when a new rule arrives from the SDN controller, *CCG* checks whether the network would satisfy the desired invariants if the rule were applied. If so, the rule is sent without delay; otherwise, it is buffered, and at each step *CCG* checks its buffer to see if any rules can be installed safely (via repeated verifications).

This simplistic algorithm has two key problems. First, the greedy algorithm may not find the best (e.g., fastest) update installation sequence, and even worse, it may get stuck with no update being installable without violating an invariant. However, we identify a fairly large scope of policies that are "segment-independent" for which the heuristic is guaranteed to succeed without deadlock (§5.2). For non-segment-independent policies, *CCG* needs a more heavyweight update technique, such as Consistent Updates [25] or SWAN [13], to act as a fallback. But *CCG* triggers this fallback mechanism only when the greedy heuristic determines it cannot offer a feasible update sequence. This is very rare in practice for the invariants we test (§7), and even when the fallback is triggered, only a small part of the transition is left to be handled by it, so the overhead associated with the heavyweight mechanism (e.g., delay and temporarily doubled FIB entries) is avoided as much as possible.

The second challenge lies in the verifier. Existing real-time data plane verifiers, such as VeriFlow and NetPlumber, assume that they have an accurate network-wide snapshot; but the network is a distributed system and we cannot know exactly when updates are applied. To address that, *CCG* explicitly models the uncertainty about network state that arises due to timing, through the use of *uncertain forwarding graph* (§4), a data structure that compactly represents the range of possible network behaviors given the available information. Although compact, *CCG*'s verification engine produces potentially larger models than those of existing tools due to this "uncertainty" awareness. Moreover, as a subroutine of the scheduling procedure, the verification function is called much more frequently than when it is used purely for verification. For these reasons, a substantial amount of work went into optimization, as shown in §7.1.

In summary, our contributions are:

- We developed a system, *CCG*, to efficiently synthesize network update orderings to preserve customizable policies as network states evolve.
- We created a graph-based model to capture network uncertainty, upon which real-time verification is performed (90% of updates verified within 10 μs).
- We evaluate the performance of our *CCG* implementation in both emulation and a physical testbed, and demonstrate that *CCG* offers significant performance improvement over previous work—up to 3× faster updates, typically with zero extra FIB entries—while preserving various levels of consistency.

2 Problem Definition and Related Work

We design *CCG* to achieve the following objectives:

1) Consistency at Every Step. Network changes can occur frequently, triggered by the control applications, changes in traffic load, system upgrades, or even failures. Even in SDNs with a logically centralized controller, the asynchronous and distributed nature implies that no single component can always obtain a fully up-to-date view of the entire system. Moreover, data packets from all possible sources may traverse the network at any time in any order, interleaving with the network data plane updates. How can we continuously enforce consistency properties, given the incomplete and uncertain network view at the controller?

2) Customizable Consistency Properties. The range of desired consistency properties of networks is quite diverse. For example, the successful operations of some networks may depend on a set of paths traversing a firewall, certain "classified" hosts being unreachable from external domains, enforcement of access control to protect critical assets, balanced load across links, loop freedom, etc. As argued in [21], a generic framework to handle general properties is needed. Researchers have attempted to ensure certain types of consistency properties, e.g., loop freedom or absence of packet loss [13,19], but those studies do not provide a generalized solution. Dionysus [15], as stated earlier, generalizes the scope of consistency properties it deals with, but still requires designing specific algorithms for different invariants. Consistent Updates [25] is probably the closest solution to support general consistency properties because it provides the relatively strong property of packet coherence which is sufficient to guarantee many other properties; but as we will see next, it sacrifices efficiency.

3) Efficient Update Installation. The network controller should react in a timely fashion to network changes to minimize the duration of performance drops and network errors. There have been proposals [13, 16, 19, 23, 25] that instill correctness according to a specific consistency property, but these approaches suffer substantial performance penalties. For example, the waiting time between phases using the two-phase update scheme proposed in CU [25] is at least the maximum delay across all the devices, assuming a completely parallel implementation. Dionysus [15] was recently proposed to update networks via dynamic scheduling atop a consistency-preserving dependency graph. However, it requires implementing a new algorithm and dependency graph for each new invariant to achieve good performance. For example, a packet coherence invariant needs one algorithm and a waypoint invariant would need another algorithm. In contrast, our approach reduces the consistency problem to a general network verification problem, which can take a broad range of invariants as inputs. In particular, one only needs to specify the verification function instead of designing a new algorithm. This approach also grants *CCG* the ability to deal with wildcard rules efficiently, in the same way as general verification tools, whereas Dionysus only works for applications with exact match on flows or classes of flows.

3 Overview of *CCG*

CCG converts the update scheduling problem into a network verification problem. Our overall approach is shown in Figure 1. Our uncertainty-aware network model (§4.2) provides a compact symbolic representation of the different possible states the network could be in, providing input for the verification engine. The verification engine is responsible for verifying application updates against specified invariants and policies (§4.4). Based on verification results, *CCG* synthesizes an efficient update plan to preserve policy consistency during network updates, using the basic heuristic and a more heavyweight fallback mechanism as backup (§5.1 and §5.3). One key feature of *CCG* is that it operates in a *black-box* fashion, providing a general platform with a very flexible notion of consistency. For example, one can "plug in" a different verification function and a fallback update scheduling tool to meet one's customized needs.

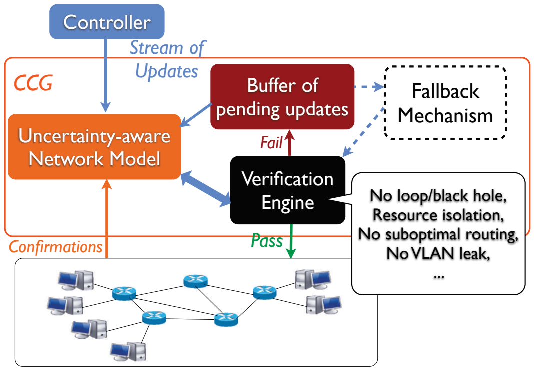


Figure 1: ***System architecture of CCG.***

4 Verification under Uncertainty

We start by describing the problem of network uncertainty (§4.1), and then present our solution to model a network in the presence of uncertainty (§4.2 and §4.3). Our design centers around the idea of *uncertain forwarding graphs*, which compactly represent the entire set of possible network states from the standpoint of packets. Next, we describe how we use our model to perform uncertainty-aware network verification (§4.4).

4.1 The Network Uncertainty Problem

Networks must disseminate state among distributed and asynchronous devices, which leads to the inherent *uncertainty* that an observation point has in knowing the current state of the network. We refer to the time period during which the view of the network from an observation point (e.g., an SDN controller) might be inconsistent with the actual network state as *temporal network uncertainty*. The uncertainty could cause network behaviors to deviate from the desired invariants temporarily or even permanently.

Figure 2 shows a motivating example. Initially, switch A has a forwarding rule directing traffic to switch B. Now the operator wants to reverse the traffic by issuing two instructions in sequence: (1) remove the rule on A, and (2) insert a new rule (directing traffic to A) on B. But it is possible that the second operation finishes earlier than the first one, causing a transient loop that leads to packet losses. That is not an uncommon situation; for example, three out of eleven bugs found by NICE [7] (BUG V, IX and XI) are caused by the control programs' lack of knowledge of the network states.

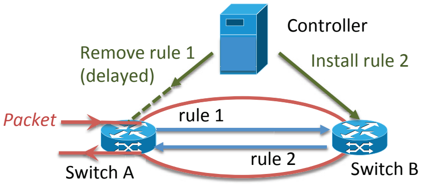


Figure 2: ***Example: challenge of modeling networks in the presence of uncertainty.***

Such errors may have serious consequences. In the previous example, the resulting packet losses could cause a significant performance drop. A recent study [9] shows TCP transfers with loss may take five times longer to complete. Other transient errors could violate security policy, e.g., malicious packets could enter a secure zone because of a temporary access control violation [25].

To make matters worse, errors caused by unawareness of network temporal uncertainty can be permanent. For instance, a control program initially instructs a switch to install one rule, and later removes that rule. The two instructions can be reordered at the switch [11], which ultimately causes the switch to install a rule that ought to be removed. The view of the controller and the network state will remain inconsistent until the rule expires. One may argue that inserting a barrier message in between the two instructions would solve the problem. However, this may harm performance because of increasing control traffic and switch operations. There are also scenarios in which carefully crafting an ordering does not help [25]. In addition, it is difficult for a controller to figure out when to insert the barrier messages. *CCG* addresses that by serializing only updates that have potential to cause race conditions that violate an invariant (§6).

4.2 Uncertainty Model

We first briefly introduce our prior work VeriFlow, a real-time network-wide data plane verifier. VeriFlow intercepts every update issued by the controller before it hits the network and verifies its effect in real time. VeriFlow first slices the set of possible packets into Equivalence Classes (ECs) of packets using all existing forwarding rules and the new update. Each EC is a set of packets that experiences the same forwarding actions throughout the network. Next, VeriFlow builds a *forwarding graph* for each EC affected by the update, by collecting forwarding rules influencing the EC. Lastly, VeriFlow traverses each of these graphs to verify network-wide invariants.

Naively, to model network uncertainty, for every update, we need two graphs to symbolically represent the network behavior with and without the effect of the update for each influenced EC, until the controller is certain about the status of the update. If n updates are concurrently "in flight" from the controller to the network, we would need 2^n graphs to represent all possible sequences of update arrivals. Such a state-space explosion will result in a huge memory requirement and excessive processing time to determine consistent update orderings.

To address that, we efficiently model the network forwarding behavior as a *uncertain forwarding graph*, whose links can be marked as *certain* or *uncertain*. A forwarding link is *uncertain* if the controller does not yet have information on whether that corresponding update has been applied to the network. The graph is maintained by the controller over time. When an update is sent, its effect is applied to the graph and marked as uncertain. After receipt of an acknowledgment from the network that an update has been applied (or after a suitable timeout), the state of the related forwarding link is changed to *certain*. Such a forwarding graph represents all possible combinations of forwarding decisions at all the devices.

In this way, the extra storage required for uncertainty modeling is linearly bounded by the number of uncertain rules. We next examine when we can resolve uncertainty, either confirming a link as certain or removing it.

4.3 Dynamic Updating of the Model

In order to model the most up-to-date network state, we need to update the model as changes happen in the network. At first glance, one might think that could be done simply by marking forwarding links as uncertain when new updates are sent, and then, when an ack is received from the network, marking them as certain. The problem with that approach is that it may result in inconsistencies from the data packets' perspective. Consider a network consisting of four switches, as in Figure 4.

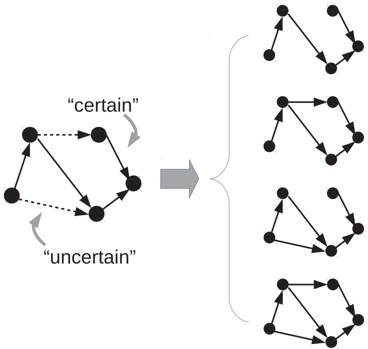


Figure 3: ***CCG's uncertain forwarding graph.***

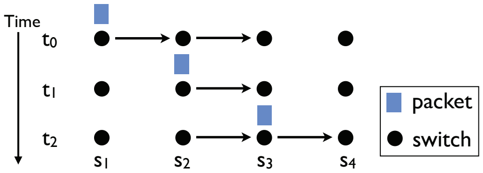


Figure 4: ***Example: challenge of dealing with non-atomicity of packet traversal.***

The policy to enforce is that packets from a particular source entering Switch s_1 should not reach Switch s_4. Initially, at time t_0, Switch s_3 has a filtering rule to drop packets from that source, whereas all the other switches simply pass packets through. The operator later wants to drop packets on s_1 instead of s_3. To perform the transition in a conservative way, the controller first adds a filtering rule on s_1 at t_1, then removes the filtering rule on s_3 at t_2, after the first rule addition has been confirmed.

The forwarding graphs at all steps seem correct. However, if a packet enters s_1 before t_1 and reaches s_3 after t_2, it will reach s_4, which violates the policy. Traversal of a packet over the network is not atomic, interleaving with network updates, as also observed in [25]. Moreover, [20] recently proved that there are situations where no correct update order exists. To deal with it, upon receiving an ack from the network, *CCG* does not immediately mark the state of the corresponding forwarding link as certain. Instead, it delays application of the confirmation to its internal data structure. In fact, confirmations of additions of forwarding links in the graph model can be processed immediately, and only confirmations of removals of forwarding links need to be delayed. The reason is that we want to ensure we represent all the possible behaviors of the network. Even after a forwarding rule has been deleted, packets processed by the rule may still exist in the network, buffered in an output queue of that device, in flight, or on other devices.

We have proved that our uncertainty-aware model is able to accurately capture the view of the network from the packets' perspective [2], even for in-flight packets that have been affected by rules not currently present.

Definition 1. *A packet P's view of the network* **agrees** *with the uncertainty-aware model, if at any time point during its traversal of the network, the data plane state that the packet encounters is in the model at that time point. More specifically, at time t, to P if a link l*

- *is reachable, l is in the graph model for P at t;*
- *otherwise, l is definitely not certain in the graph at t.*

Theorem 1. *Assuming that all data plane changes are initiated by the controller, any packet's view of the network agrees with the uncertainty-aware model.*

4.4 Uncertainty-aware Verification

Construction of a *correct* network verification tool is straightforward with our uncertainty-aware model. By traversing the uncertainty graph model using directed graph algorithms, we can answer queries such as whether a reachable path exists between a pair of nodes. That can be done in a manner similar to existing network verification tools like HSA [17] and VeriFlow [18]. However, the traversal process needs to be modified to take into account uncertainty. When traversing an uncertain link, we need to keep track of the fact that downstream inferences lack certainty. If we reach a node with no *certain* outgoing links, it is possible that packets will encounter a black-hole even with multiple *uncertain* outgoing links available. By traversing the graph once, *CCG* can reason about the network state correctly in the presence of uncertainty, determine if an invariant is violated, and output the set of possible conterexamples (e.g., a packet and the forwarding table entries that caused the problem).

5 Consistency under Uncertainty

In this section, we describe how we use our model to efficiently synthesize update sequences that obey a set of provided invariants (§5.1). We then identify a class of invariants that can be guaranteed in this manner (§5.2), and present our technique to preserve consistency for broader types of invariants (§5.3).

5.1 Enforcing Correctness with Greedily Maximized Parallelism

The key goal of our system is to instill user-specified notions of correctness during network transitions. The basic idea is relatively straightforward. We construct a *buffer* of updates received from the application, and attempt to send them out in FIFO order. Before each update is sent, we check with the verification engine on whether there is any possibility, given the uncertainty in network state, that sending it could result in an invariant violation. If so, the update remains buffered until it is safe to be sent.

There are two key problems with this approach. The first is head-of-line blocking: it may be safe to send an update, but one before it in the queue, which isn't safe, could block it. This introduces additional delays in propagating updates. Second, only one update is sent at a time, which is wasteful—if groups of updates do not conflict with each other, they could be sent in parallel.

To address this, *CCG* provides an algorithm for synthesizing update sequences to networks that greedily *maximizes parallelism* while simultaneously obeying the supplied properties (Algorithm 1).

Whenever an update u is issued from the controller, *CCG* intercepts it before it hits the network. Network forwarding behavior is modeled as an uncertainty graph ($G_{uncertain}$) as described previously. Next, the black-box verification engine takes the graph and the new update as input, and performs a computation to determine whether there is any possibility that the update will cause the graph state to violate any policy internally specified within this engine. If the verification is passed, the update u is sent to the network and also applied to the network model $Model$, but marked as uncertain. Otherwise, the update is buffered temporarily in Buf.

When a confirmation of u from the network arrives, *CCG* also intercepts it. The status of u in $Model$ is changed to certain, either immediately (if u doesn't remove any forwarding link from the graph), or after a delay (if it does, as described in §4.3). The status change of u may allow some pending updates that previously failed the verification to pass it. Each of the buffered updates is processed through the routine of processing a new update, as described above.

In this way, *CCG* maintains the order of updates only when it matters. Take the example in Figure 2. If the deletion of rule 1 is issued before the addition of rule 2 is confirmed, *CCG*'s verification engine will capture a possible loop, and thus will buffer the deletion update. Once the confirmation of adding rule 2 arrives, *CCG* checks buffered updates, and finds out that now it's safe to issue the deletion instruction.

5.2 Segment Independence

Next, we identify a class of invariants for which a feasible update ordering exists, and for which *CCG*'s heuristic will be guaranteed to find one such order. As defined in [25], *trace properties* characterize the paths that packets traverse through the network. This covers many common network properties, including reachability, access control, loop freedom, and waypointing. We start with the assumption that a network configuration applies to exactly one equivalence class of packets. A network configuration can be expressed as a set of paths that packets are allowed to take, i.e., a forwarding graph. A configuration transition is equivalent to a transition from an initial forwarding graph, G_0, to a final graph, G_f, through a series of transient graphs, G_t, for $t \in \{1,\dots,f-1\}$. We assume throughout that the invariant of interest is preserved in G_0 and G_f.

Algorithm 1 Maximizing network update parallelism

```
ScheduleIndividualUpdate($Model, Buf, u$)
  On issuing $u$:
  $G_{uncertain}$ = ExtractGraph($Model, u$)
  $verify$ = BlackboxVerification($G_{uncertain}, u$)
  if $verify$ == PASS then
      Issue $u$
      Update($Model, u, uncertain$)
  else
      Buffer $u$ in $Buf$

  On confirming $u$:
  Update($Model, u, certain$)
  $Issue\_updates \leftarrow \emptyset$
  for $u_b \in Buf$ do
      $G_{uncertain}$ = ExtractGraph($Model, u_b$)
      $verify$ = BlackboxVerification($G_{uncertain}, u_b$)
      if $verify$ == PASS then
          Remove $u_b$ from $Buf$
          Update($Model, u_b, uncertain$)
          $Issue\_updates \leftarrow Issue\_updates + u_b$
  Issue $Issue\_updates$
```

Loop and black-hole freedom The following theorems were proved for loop freedom [10]: First, given both G_0 and G_f are loop-free, during transition, it is safe (causing no loop) to update a node in any G_t, if that node satisfies one of the following two conditions: (1) in G_t it is a leaf node, or all its upstream nodes have been updated with respect to G_f; or (2) in G_f it reaches the destination directly, or all its downstream nodes in G_f have been updated with respect to G_f. Second, if there are several updatable nodes in a G_t, any update order among these nodes is loop-free. Third, in any loop-free G_t (including G_0) that is not G_f, there is at least one node safe to update, i.e., a loop-free update order always exists.

Similarly, we have the following proved for the black-hole freedom property [2].

Lemma 1. *(Updatable condition): A node update does not cause any transient black-hole, if in G_f, the node reaches the destination directly, or in G_t, all its downstream nodes in G_f have already been updated.*

Proof. By contradiction. Let $N_0, N_1,...N_n$ be downstream nodes of N_a in G_f. Assume N_0, N_1,...N_n have been updated with respect to G_f in G_t. After updating N_a in G_t, N_0, N_1,...N_n become N_a's downstream nodes and all nodes in the chain from N_a to N_n have been updated. N_a's upstream with respect to G_t can still reach N_a, and thus reach the downstream of N_a. If we assume there is a black-hole from updating N_a, there exists a black-hole in the chain from N_a to N_n. Therefore, the black-hole will exist in G_f, and there is a contradiction. □

Lemma 2. *(Simultaneous updates): Starting with any G_t, any update order among updatable nodes is black-hole-free.*

Proof. Consider a updatable node N_a such that all its downstream nodes in G_f have already been updated in G_t (Lemma 1). Then updating any other updatable node does not change this property. When a node is updatable it remains updatable even after updating other nodes. Therefore, if there are several updatable nodes, they can be updated in any order or simultaneously. □

Theorem 2. *(Existence of a black-hole-free update order): In any black-hole-free G_t that is not G_f (including G_0), at least one of the nodes is updatable, i.e., there is a black-hole-free update order.*

Proof. By contradiction. Assume there is a transient graph G_t such that no node is updatable. All nodes are either updated or not updatable. As nodes with direct links to the destination are updatable (Lemma 1), these nodes can only be updated. Then nodes at previous hop of these nodes in G_t are also updatable (Lemma 1), and therefore these nodes must also be updated. Continuing, it follows that all nodes are updated, which is a contradiction as G_t = G_f . As there is always a node updatable in a consistent G_t, and the updatable node can be updated to form a new consistent G_t, the number of updated nodes will increase. Eventually, all nodes will be updated. Therefore there is a black-hole free update order. □

Any update approved by *CCG* results in a consistent transient graph, so *CCG* always finds a consistent update sequence to ensure loop and black-hole freedom.

Generalized Trace Properties To get a uniform abstraction for trace properties, let us first visit the basic connectivity problem: node A should reach node B ($A \rightarrow B$). To make sure there is connectivity between two nodes, both black-hole and loop freedom properties need to hold. Obviously, black-hole freedom is downstream-dependent (Theorem 2), whereas loop freedom is upstream- (updatable condition (1)) **or** downstream-dependent (updatable condition (2)), and thus weaker than black-hole freedom. In other words, connectivity is a downstream-dependent property, i.e., updating from downstream to upstream is sufficient to ensure it. Fortunately, a number of trace properties, such as waypointing, access control, service/middle box chaining, etc., can be broken down to basic connectivity problems. A common characteristic of such properties is that flows are required to traverse a set of waypoints.

Definition 2. Waypoints-based trace property: *A property that specifies that each packet should traverse a set of waypoints (including source and destination) in a particular order.*

Definition 3. Segment dependency: *Suppose a trace property specifies n waypoints, which divide the old and the new flow path each into $(n-1)$ segments: $old_1, old_2, ..., old_{n-1}$ and $new_1, new_2, ..., new_{n-1}$. If new_j*

crosses old_i ($i \neq j$), then the update of segment j is **dependent** *on the update of segment i, i.e., segment j cannot start to update until segment i's update has finished, in order to ensure the traversal of all waypoints.*

Otherwise, if segment j starts to update before i has finished, there might be violations. If $j < i$, there might be a moment when the path between waypoints j and $i+1$ consists only of new_j and part of old_i, i.e., waypoints $(j+1)...i$ are skipped. As in Figure 5(b), B may be skipped if the AB segment is updated before BC, and the path is temporarily $A \rightarrow 2 \rightarrow C$.

If $j > i$, there might be a moment when the path between waypoints i and $(j+1)$ consists of $old_i, old_{i+1}, ..., new_j$, and a loop is formed. As in Figure 5(c), the path could temporarily be $A \rightarrow B \rightarrow 1 \rightarrow B$.

If there is no dependency among segments (Figure 5 (a)), then each can be updated independently simply by ensuring connectivity between the segment's endpoints. That suggests that for paths with no inter-segment dependencies, a property-compliant update order always exists. Another special case is circular dependency between segments, as depicted in Figure 5(d), in which no feasible update order exists.

Theorem 3. *If there is no circular dependency between segments, then an update order that preserves the required property always exists. In particular, if policies are enforcing no more than two waypoints, an update order always exists.*

If a policy introduces no circular dependency, i.e., at least one segment can be updated independently (Figure 5(a-c)), then we say the policy is *segment independent*. However, in reality, forwarding links and paths may be shared by different sets of packets, e.g., multiple flows. Thus it is possible that two forwarding links (smallest possible segments) l_1 and l_2 will have conflicting dependencies when serving different groups of packets, e.g., in forwarding graphs destined to two different IP prefixes. In such cases, circular dependencies are formed across forwarding graphs. Fortunately, forwarding graphs do not share links in many cases. For example, as pointed out in [15], a number of flow-based traffic management applications for the network core (e.g., ElasticTree, MicroTE, B4, SWAN [6, 12–14]), any forwarding rule at a switch matches at most one flow.

Other Properties There are trace properties which are not waypoint-based, such as quantitative properties like path length constraint. To preserve such properties and waypoint-based trace properties that are not segment independent, we can use other heavyweight techniques as a fallback (see 5.3), such as CU [25]. Besides, there are network properties beyond trace properties, such as congestion freedom, and it has been proven that careful ordering of updates cannot always guarantee congestion freedom [13, 27]. To ensure congestion freedom, one approach is to use other heavyweight tools, such as SWAN [13], as a fallback mechanism that the default heuristic algorithm can trigger only when necessary.

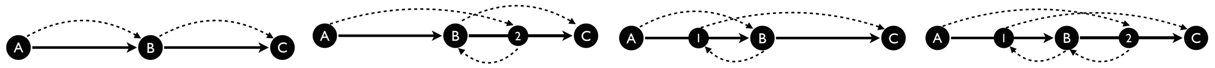
5.3 Synthesis of Consistent Update Schedules

When desired policies do not have the segment-independence property (§5.2), it is possible that some buffered updates (through very rare in our experiments) never pass the verification. For instance, consider a circular network with three nodes, in which each node has two types of rules: one type to forward packets to destinations directly connected to itself, and one default rule, which covers destinations connected to the other two switches. Initially, default rules point clockwise. They later change to point counterclockwise. No matter which of the new default rules changes first, a loop is immediately caused for some destination. The loop freedom property is not segment-independent in this case, because each default rule is shared by two equivalence classes (destined to two hosts), which results in conflicting dependencies among forwarding links.

To handle such scenarios, we adopt a hybrid approach (Algorithm 2). If the network operators desire some policies that can be guaranteed by existing solutions, e.g., CU or SWAN, such solutions can be specified and plugged in as the fallback mechanism, FB. The stream of updates is first handled by *CCG*'s greedy heuristic (Algorithm 1) as long as the policy is preserved. Updates that violate the policy are buffered temporarily. When the buffering time is over threshold T, configured by the operator, the fallback mechanism is triggered. The remaining updates are fed into FB to be transformed to a feasible sequence, and then Algorithm 1 proceeds with them again to heuristically maximize update parallelism. In that way, *CCG* can always generate a consistent update sequence, *assuming a fallback mechanism exists which can guarantee the desired invariants*.[1] Note that even with FB triggered, *CCG* achieves better efficiency than using FB alone to update the network, because: 1) in the common case, most of updates are not handled by FB; 2) *CCG* only uses FB to "translate" buffered updates and then heuristically parallelize issuing the output of FB, but doesn't wait explicitly as some FB mechanism does, e.g., the waiting time between two phases in CU.

To show the feasibility of that approach, we implemented both CU [25] (see §7) and SWAN [13] as our fallback mechanisms in *CCG*. We emulated traffic engi-

[1] If no appropriate fallback exists, and the invariant is non-segment-independent, *CCG* can no longer guarantee the invariant. In this case, *CCG* can offer a "best effort" mechanism to maintain consistency during updates by simply releasing buffered updates to the network after a configurable threshold of time. This approach might even be preferable for certain invariants where operators highly value update efficiency; we leave an evaluation to future work.



(a) No segment crossing, update different segments in parallel, as long as each segment's updating follows downstream dependency

(b) Old path: $A \rightarrow B \rightarrow 2 \rightarrow C$, new path: $A \rightarrow 2 \rightarrow B \rightarrow C$. New *AB* crosses old *BC*, so *AB* depends on *BC*

(c) Old path: $A \rightarrow 1 \rightarrow B \rightarrow C$, new path: $A \rightarrow B \rightarrow 1 \rightarrow C$. New *BC* crosses old *AB*, so *BC* depends on *AB*

(d) Old path: $A \rightarrow\rightarrow 1 \rightarrow B \rightarrow 2 \rightarrow C$, new path: $A \rightarrow 2 \rightarrow B \rightarrow 1 \rightarrow C$. New *BC* crosses old *AB*, and new *AB* crosses old *BC*, so *BC* and *AB* have circular dependency between themselves.

Figure 5: ***Examples: dependencies between segments. Path** AC **is divided into two segments** AB **and** BC **by three waypoints** A, B, **and** C, **with old paths in solid lines, and new paths in dashed lines.***

Algorithm 2 Synthesizing update orderings

```
ScheduleUpdates($Model, Buf, U, FB, T$)
  for $u \in U$ do
    ScheduleIndividualUpdate($Model, Buf, u$)

  On timeout($T$):
  $\tilde{U}$ = Translate($Buf, FB$)
  for $u \in \tilde{U}$ do
    ScheduleIndividualUpdate($Model, Buf, u$)
```

neering (TE) and failure recovery (FR), similar to Dionysus [15], in the network shown in Figure 6. Network updates were synthesized to preserve congestion-freeness using *CCG* (with SWAN as plug-in), and for comparison, using SWAN alone. In the TE case, we changed the network traffic to trigger new routing updates to match the traffic. In the FR case, we turned down the link S3-S8 so that link S1-S8 was overloaded. Then the FR application computed new updates to balance the traffic. The detailed events that occurred at all eight switches are depicted in Figure 7. We see that *CCG* ensured the same consistency level, but greatly enhanced parallelism, and thus achieved significant speed improvement (1.95× faster in the TE case, and 1.97× faster in the FR case).

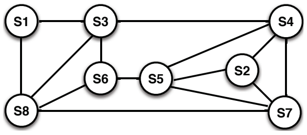


Figure 6: ***Topology for CCG and SWAN bandwidth tests***

6 Implementation

We implemented a prototype of *CCG* with 8000+ lines of C++ code. *CCG* is a shim layer between an SDN controller and network devices, intercepting and scheduling network updates issued by the controller in real time.

CCG maintains several types of state, including network-wide data plane rules, the uncertainty state of each rule, the set of buffered updates, and bandwidth information (e.g., for congestion-free invariants). It stores data plane rules within a multi-layer trie in which each layer's sub-trie represents a packet header field. We designed a customized trie data structure for handling different types of rule wildcards, e.g., full wildcard, subnet wildcard, or bitmask wildcard [24], and a fast one-pass traversal algorithm to accelerate verification. To handle wildcarding for bitmasks, each node in the trie has three child branches, one for each of {0, 1, don't care}. For subnetting, the wildcard branch has no children, but points directly to a next layer sub-trie or a rule set. Thus, unlike other types of trie, the depth of subnet wildcard tries is not fixed as the number of bits in this field, but instead equals to the longest prefix among all the rules it stores. Accordingly, traversal cost is reduced compared with general tries. For full wildcard fields, values can only be non-wildcarded or full wildcarded. The specialized trie structure for this type of field is a plain binary tree plus a wildcard table.

When a new update arrives, we need to determine the set of affected ECs, as well as the rules affecting each EC. VeriFlow [18] performs a similar task via a two-pass algorithm, first traversing the trie to compute a set of ECs, and then for each of the discovered ECs, traversing the trie again to extract related rules. In *CCG*, using callback functions and depth first searching, the modeling work is finished with only one traversal. This algorithm eliminates both the unnecessary extra pass over the trie and the need to allocate memory for intermediate results.

In addition to forwarding rules, the data structure and algorithm are also capable of handling packet transformation rules, such as Network Address Translation (NAT) rules, and rules with VLAN tagging, which are used by CU for versioning, and verified by *CCG* when the CU plug-in is triggered (see §7).

To keep track of the uncertainty states of rules, we design a compact state machine, which enables *CCG* to detect rules that cause potential race conditions. If desired, our implementation can be configured to insert barrier messages to serialize those rule updates.

To bound the amount of time that the controller is uncertain about network states, we implemented two alternate types of the confirmation mechanisms: (1) an application-level acknowledgment by modifying the user-space switch program in Mininet, and (2) leveraging the barrier and barrier reply messages for our physical SDN testbed experiments.

CCG exposes a set of APIs that can be used to write general queries in C++. The APIs allow the network operator to get a list of affected equivalence classes given an arbitrary forwarding rule, the corresponding forwarding graphs, as well as traverse these graphs in a controlled manner and check properties of interest. For instance, an operator can ensure packets from an insecure source

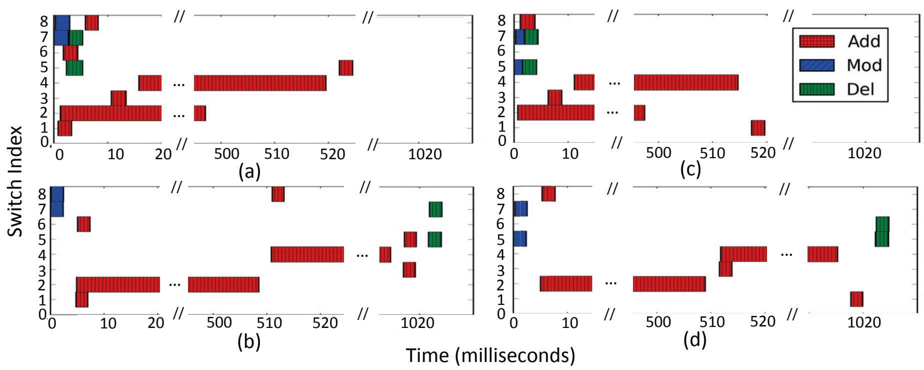


Figure 7: ***Time series of events that occurred across all switches: (a) SWAN + CCG, traffic engineering; (b) SWAN, traffic engineering; (c) SWAN + CCG, failure recovery; (d) SWAN, failure recovery. In both cases, CCG + SWAN finishes about 2x faster.***

encounter a firewall before accessing an internal server.

7 Evaluation

7.1 Verification Time

To gain a baseline understanding of *CCG*'s performance, we micro-benchmarked how long the verification engine takes to verify a single update. We simulated BGP routing changes by replaying traces collected from the Route Views Project [4], on a network consisting of 172 routers following a Rocketfuel topology (AS 1755) [1]. After initializing the network with 90,000 BGP updates, 2,559,251 updates were fed into *CCG* and VeriFlow [18] (as comparison). We also varied the number of concurrent uncertain rules in *CCG* from 100 to 10,000. All experiments were performed on a 12-core machine with Intel Core i7 CPU at 3.33 GHz, and 18 GB of RAM, running 64-bit Ubuntu Linux 12.04. The CDFs of the update verification time are shown in Figure 8.

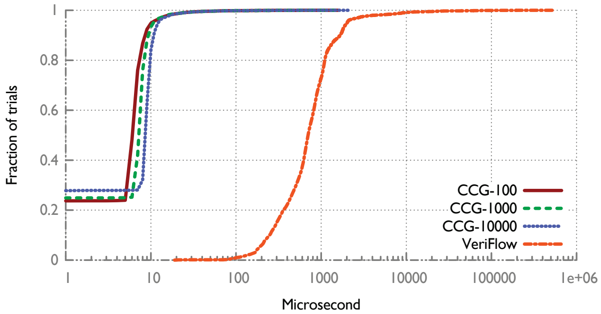


Figure 8: ***Microbenchmark results.***

CCG was able to verify 80% of the updates within 10 μs, with a 9 μs mean. *CCG* verifies updates almost two order of magnitude faster than VeriFlow because of data structure optimizations (§6). Approximately 25% of the updates were processed within 1 μs, because *CCG* accurately tracks the state of each rule over time. When a new update matches the pattern of some existing rule, it's likely only a minimum change to *CCG*'s network model is required (e.g., only one operation in the trie, with no unnecessary verification triggered). We observed long tails in all curves, but the verification time of *CCG* is bounded by 2.16 ms, almost three orders of magnitude faster than VeriFlow's worst case. The results also show strong scalability. As the number of concurrent uncertainty rules grows, the verification time increases slightly (on average, 6.6 μs, 7.3 μs, and 8.2 μs for the 100-, 1000-, and 10000-uncertain-rule cases, respectively). Moreover, *CCG* offers a significant memory overhead reduction relative to VeriFlow: 540 MB vs 9 GB.

7.2 Update Performance Analysis

7.2.1 Emulation-based Evaluation

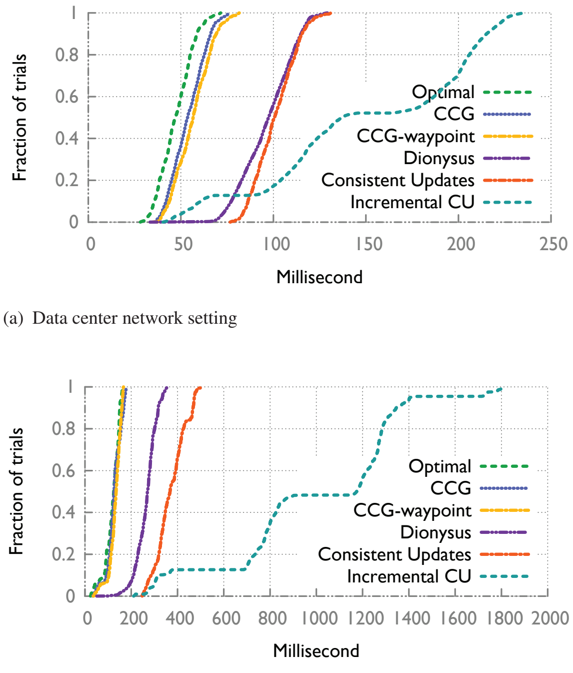
Segment-independent Policies: We used Mininet to emulate a fat-tree network with a shortest path routing application and a load-balancing application in a NOX controller. The network consists of five core switches and ten edge switches, and each edge switch connects to five hosts. We change the network (e.g., add links, or migrate hosts) to trigger the controller to update the data plane with a set of new updates. For each set of experiments, we tested six update mechanisms: (1) the controller immediately issues updates to the network, which is **Optimal** in terms of update speed; (2) *CCG* with the basic connectivity invariants, loop and black-hole freedom, enabled (*CCG*); (3) *CCG* with an additional invariant that packets must traverse a specific middle hop before reaching the destination (*CCG*-waypoint); (4) Consistent Updates (**CU**) [25]; (5) incremental Consistent Updates (**Incremental CU**) [16]; and (6) **Dionysus** [15] with its WCMP forwarding dependency graph generator. We configure our applications as the same type as in Dionysus, with forwarding rules matching exactly one flow, i.e., no overlapping forwarding graphs. Thus, loop and black-hole freedom are segment-independent as proved in §5.2. Because of the fat-tree structure, there is no crossing between path segments (as in Fig 5(a)), so the waypoint policy is also segment independent. A mix of old and new configurations, e.g., $oldAB + newBC$ in Figure 5(a), is allowed by *CCG*, but forbidden when using CU. Note here, we used our own implementation of the algorithms introduced in Dionysus paper, specifically the algorithm for packet coherence. Therefore, this is not a full evaluation of the Dionysus *approach*: one can develop special-purpose algorithms that build customized dependency graphs for weaker properties, and thus achieve better efficiency. We leave such evaluation to future work.

We first set the delay between the controller issuing an update and the corresponding switch finishing the application of the update (i.e, the controller-switch delay) to a normal distribution with 4 ms mean and 3 ms jitter, to mimic a dynamic data center network environment. The settings are in line with that of other data center SDN experiments [8, 26]. We initialized the test with one core switch enabled and added the other four core switches after 10 seconds. The traffic eventually is evenly distributed across all links because of the load balancer application. We measured the completion time of updating each communication path, repeated each experiment 10 times. Figure 9(a) shows the CDFs for all six scenarios.

The performance of both "*CCG*" and "*CCG*-waypoint" is close to optimal, and much faster (47 ms reduction on average) than CU. In CU, the controller is required to wait for the maximum controller-switch delay to guarantee that all packets can only be handled by either the old or the new rules. *CCG* relaxes the constraints by allowing a packet being handled by a mixture of old and new rules along the paths, as long as the impact of the new rules passed verification. By doing so, *CCG* can apply any verified updates without explicitly waiting for irrelevant updates. CU requires temporary doubling of the FIB space for each update, because it does not delete old rules until all in-flight packets processed by the old configuration have drained out of the network. To address this, incremental-CU was proposed to trade time against flow table space. By breaking a batch of updates into k subgroups ($k = 3$ in our tests), incremental-CU reduced the extra memory usage to roughly one kth at the cost of multiplying the update time k times. In contrast, when dealing with segment-independent policies, as in this set of experiments, *CCG* never needs to trigger any heavyweight fallback plug-in, and thus requires no additional memory, which is particularly useful as switch TCAM memory can be expensive and power-hungry.

To understand how *CCG* performs in wide-area networks, where SDNs have also been used [13, 14], we set the controller-switch delay to 100 ms (normal distribution, with 25ms jitter), and repeated the same tests (Figure 9(b)). *CCG* saved over 200 ms update completion time compared to CU, mainly due to the longer controller-switch delay, for which CU and incremental-CU have to wait between the two phases of updates.

As for Dionysus, we observed in Figure 9 that it speeds up updates compared to CU in both local and wide-area settings, as it reacts to network dynamics rather than pre-determining a schedule. But because its default algorithm for WCMP forwarding produces basically the same number of updates as CU, *CCG* (either *CCG* or *CCG*-waypoint) outperforms it in both time and memory cost. We further compared *CCG*-waypoint with Dionysus in other dynamic situations, by varying controller-switch delay distribution. Figure 10 shows the 50^{th}, 90^{th} and 99^{th} percentile update completion time, under various controller-switch delays (normal distributed with different (mean, jitter) pairs, (a,b)) for four update mechanisms: optimal, *CCG*, Dionysus, and CU. In most cases, both *CCG* and Dionysus outperform CU, with one exception (4ms delay, zero jitter). Here, Dionysus does not outperform CU because it adjusts its schedule according to network dynamics, which was almost absent in this scenario. The cost of updating dependency graphs in this scenario is relatively large compared to the small network delay. When the mean delay was larger (100ms), even with no jitter, Dionysus managed to speed the transition by updating each forwarding path independently. On the other hand, *CCG*'s performance is closer to Optimal than Dionysus. For example, in the $(4,0)$ case, *CCG* is 37%, 38%, and 52% faster than Dionysus in the 50^{th}, 90^{th} and 99^{th} percentile, respectively; in the $(100,25)$ case, *CCG* is 50%, 50%, and 53% faster than Dionysus in the 50^{th}, 90^{th} and 99^{th} percentile, respectively. Also, we observe that Dionysus's performance is highly dependent on the variance of the controller-switch delay (the larger the jitter is, the faster the update speed) because of the dynamic scheduling, but *CCG*'s performance is insensitive to the jitter.



(a) Data center network setting

(b) Wide-area network setting

Figure 9: ***Emulation results: update completion time comparison.***

Non-segment-independent Policies: We then explored

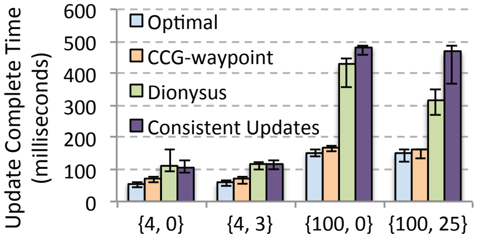


Figure 10: ***Update completion time with*** *[*50^{th}*,* 90^{th}*,* 99^{th} ***percentile]; x-axis label*** $\{a, b\}$***: a is the mean controller-switch delay, b is the jitter following a normal distribution.***

scenarios in which *CCG*'s lightweight heuristic cannot always synthesize a correct update ordering and needs to fall back to the more heavyweight algorithm to guarantee consistency. The traces we used were collected from a relatively large enterprise network that consists of over 200 layer-3 devices. During a one-day period (from 16:00 7/22/2014 to 16:00 7/23/2014), we took one snapshot of the network per hour, and used Mininet to emulate 24 transitions, each between two successive snapshots. We processed the network updates with three mechanisms: immediate application of updates, *CCG*, and CU. Updates were issued such that new rules were added first, then old rules deleted. Thus, all three mechanisms experience the trend that the number of stored rules increases then decreases.. The controller-switch delay was set to 4 ms. We selected 10 strongly connected devices in the network, and plotted the number of rules in the network over time during four transition windows, as shown in Figure 11. As the collected rules overlapped with longest prefix match, the resulting forwarding graphs might share links, so unlike previous experiments, segment-independency was not guaranteed.

The update completion time (indicated by the width of the span of each curve) using *CCG* was much shorter than CU, and the memory needed to store the rules was much smaller. In fact, the speed and memory requirements of *CCG* were close to those of the immediate update case, because *CCG* rarely needs to fall back to CU. In 22 out of 24 windows, there was a relatively small number of network updates (around 100+), much as in the [22:00, 23:00) window shown in Figure 11, in which *CCG* passed through most of the updates with very few fallbacks. During the period 23:00 to 1:00, there was a burst of network dynamics (likely to have been caused by network maintenance), in which 8000+ network updates occurred. Even for such a large number of updates, the number of updates forced to a fallback to CU, was still quite small (10+). Since *CCG* only schedules updates in a heuristic way, the waiting time of a buffered update could be suboptimal, as in this hour's case, where the final completion time of *CCG* was closer to CU. *CCG* achieves performance comparable to the immediate update mechanism, but without any of its short-term network faults (24 errors in the 0:00 to 2:00 period).

7.2.2 Physical-testbed-based Evaluation

We also evaluated *CCG* on a physical SDN testbed [3] consisting of 176 server ports and 676 switch ports, using Pica8 Pronto 3290 switches via TAM Networks, NIAGARA 32066 NICs from Interface Masters, and servers from Dell. We compared the performance of *CCG* and CU by monitoring the traffic throughput during network transitions. We first created a network with two sender-receiver pairs transmitting TCP traffic on gigabit links, shown in Figure 12. Initially, a single link was shared by the pairs, and two flows competed for bandwidth. After 90 seconds, another path was added (the upper portion with dashed lines in Figure 12). Eventually, one flow was migrated to the new path and each link was saturated. We repeated the experiment 10 times, and recorded the average throughput in a 100-ms window during the network changes. We observed repeatable results. Figure 13(a) shows the aggregated throughput over time for one trial.

CCG took 0.3 seconds less to finish the transition than CU because: (1) unlike CU, *CCG* does not require packet modification to support versioning, which takes on the order of microseconds for gigabit links, while packet forwarding is on the order of nanoseconds; (2) CU requires more rule updates and storage than *CCG*, and the speed of rule installation is around 200 flows per second; and (3) Pica8 OpenFlow switches (with firmware 1.6) cannot simultaneously process rule installations and packets.[2]

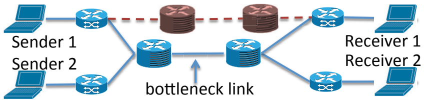


Figure 12: ***eight-switch topology.***

To test *CCG* in a larger setting, we then utilized all 13 physical switches. Each physical switch was devided into 6 "virtual" switches by creating 6 bridges. Due to the fact that the switches are physically randomly connected, this division results in a "pseudo-random" network consisting of 78 switches, each with 8 ports. Initially, the topology consisted of 60 switches, and we randomly selected 10 sender-receiver pairs to transmit TCP traffic. After 90 seconds, we enabled the remaining 18 switches in the network. The topology change triggered installations of new rules to balance load. We repeated the experiments 10 times, and selected two flows from one trial that experienced throughput changes (Figure 13(b)). The trend of the two flows is consistent with the overall observed throughput change.

CCG again outperformed CU in convergence time and average throughput during transitions. Compared to CU, *CCG* spent 20 fewer seconds to complete the transition (a reduction of 2/3), because CU waits for confirmation

[2] All the performance specifications reported in this paper have been confirmed with the Pica8 technical team.

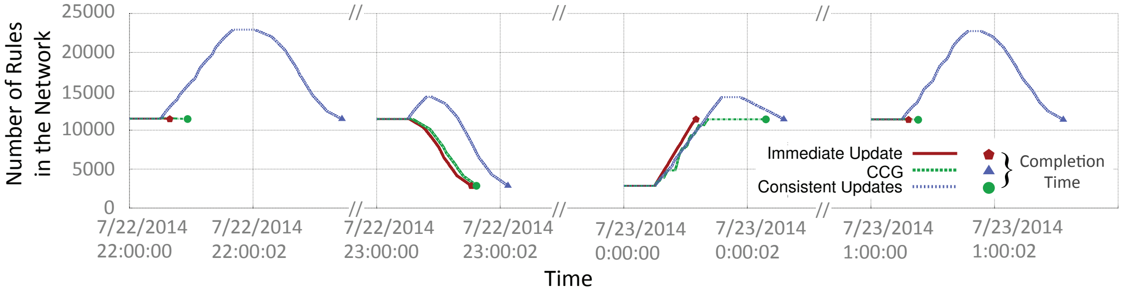
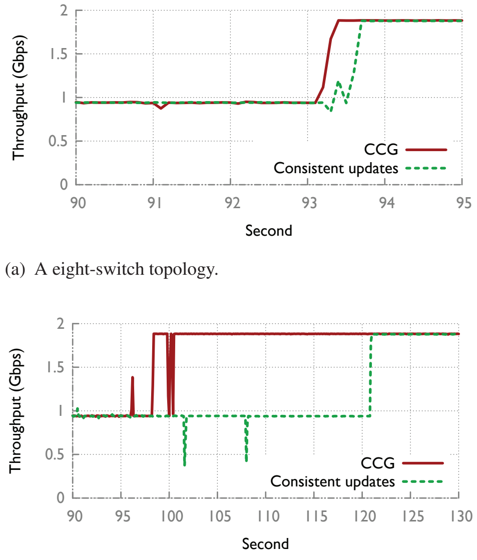


Figure 11: ***Network-trace-driven emulations: (1) immediate application of updates; (2) CCG (with CU as fallback); and (3) CU.***



(a) A eight-switch topology.

(b) A 78-switch network.

Figure 13: ***Physical testbed results: comparison of throughput changes during network transitions for CCG and CU.***

of all updates in the first phase before proceeding to the second. In contrast, *CCG*'s algorithm significantly shortened the delay, especially for networks experiencing a large number of state changes. In *CCG*, the throughput never dropped below 0.9 Gb/s, while CU experienced temporary yet significant drops during the transition, primarily due to the switches' lack of support for simultaneous application of updates and processing of packets.

8 Discussion

Limitations: *CCG* synthesizes network updates with only heuristically maximized parallelism, and in the cases where required properties are not *segment independent*, relies on heavier weight fallback mechanisms to guarantee consistency. When two or more updates have circular dependencies with respect to the consistency properties, fallback will be triggered. One safe way of using *CCG* is to provide it with a strong fallback plugin, e.g., CU [25]. Any weaker properties will be automatically ensured by *CCG*, with fallback triggered (rare in practice) only for a subset of updates and when necessary. In fact, one can use *CCG* even when fallback is always on. In this case, *CCG* will be faster most of the time, as discussed in §5.3.

Related work: Among the related approaches, four warrant further discussion. Most closely related to our work is Dionysus [15], a dependency-graph based approach that achieves a goal similar to ours. As discussed in §2, our approach has the ability to support 1) flexible properties with high efficiency without the need to implement new algorithms, and 2) applications with wildcarded rules. [22] also plans updates in advance, but using model checking. It, however, does not account for the unpredictable time switches take to perform updates. In our implementation, CU [25] and VeriFlow [18] are chosen as the fallback mechanism and verification engine. Nevertheless, they are replaceable components of the design. For instance, when congestion freedom is the property of interest, we can replace CU with SWAN [13].

Future work: We plan to study the generality of *segment independent* properties both theoretically and practically, test *CCG* with more data traces, and extend its model to handle changes initiated from the network. As comparison, we will test *CCG* against the original implementation of Dionysus with dependency graphs customized to properties of interest. We will also investigate utilizing possible primitives in network hardware to facilitate consistent updates.

9 Conclusion

We present *CCG*, a system that enforces customizable network consistency properties with high efficiency. We highlight the network uncertainty problem and its ramifications, and propose a network modeling technique correctly derives consistent outputs even in the presence of uncertainty. The core algorithm of *CCG* leverages the uncertainty-aware network model, and synthesizes a feasible network update plan (ordering and timing of control messages). In addition to ensuring that there are no violations of consistency requirements, *CCG* also tries to maximize update parallelism, subject to the constraints imposed by the requirements. Through emulations and experiments on an SDN testbed, we show that *CCG* is capable of achieving a better consistency vs. efficiency trade-off than existing mechanisms.

We thank our shepherd, Katerina Argyraki, for helpful comments, and the support of the Maryland Procurement Office under Contract No. H98230-14-C-0141.

References

[1] Rocketfuel: An ISP topology mapping engine. http://www.cs.washington.edu/research/networking/rocketfuel/.

[2] Tech report. http://web.engr.illinois.edu/~wzhou10/gcc_tr.pdf.

[3] University of illinois ocean testbed. http://ocean.cs.illinois.edu/.

[4] University of Oregon Route Views Project. http://www.routeviews.org/.

[5] E. Al-Shaer and S. Al-Haj. FlowChecker: Configuration analysis and verification of federated OpenFlow infrastructures. In *SafeConfig*, 2010.

[6] T. Benson, A. Anand, A. Akella, and M. Zhang. Microte: Fine grained traffic engineering for data centers. *CoNEXT*, 2011.

[7] M. Canini, D. Venzano, P. Peresini, D. Kostic, and J. Rexford. A NICE way to test OpenFlow applications. In *NSDI*, 2012.

[8] A. R. Curtis, J. C. Mogul, J. Tourrilhes, P. Yalagandula, P. Sharma, and S. Banerjee. DevoFlow: Scaling flow management for high-performance networks. In *ACM SIGCOMM Computer Communication Review*, volume 41, pages 254–265. ACM, 2011.

[9] T. Flach, N. Dukkipati, A. Terzis, B. Raghavan, N. Cardwell, Y. Cheng, A. Jain, S. Hao, E. Katz-Bassett, and R. Govindan. Reducing web latency: the virtue of gentle aggression. In *SIGCOMM*, 2013.

[10] J. Fu, P. Sjodin, and G. Karlsson. Loop-free updates of forwarding tables. *IEEE Transactions on Network and Service Management*, March 2008.

[11] A. Guha, M. Reitblatt, and N. Foster. Machine-verified network controllers. *Programming Languages Design and Implementation*, 2013.

[12] B. Heller, S. Seetharaman, P. Mahadevan, Y. Yiakoumis, P. Sharma, S. Banerjee, and N. McKeown. ElasticTree: Saving energy in data center networks. In *NSDI*, volume 3, pages 19–21, 2010.

[13] C.-Y. Hong, S. Kandula, R. Mahajan, M. Zhang, V. Gill, M. Nanduri, and R. Wattenhofer. Achieving high utilization with software-driven WAN. *ACM SIGCOMM*, 2013.

[14] S. Jain, A. Kumar, S. Mandal, J. Ong, L. Poutievski, A. Singh, S. Venkata, J. Wanderer, J. Zhou, M. Zhu, et al. B4: Experience with a globally-deployed software defined WAN. In *ACM SIGCOMM*, pages 3–14. ACM, 2013.

[15] X. Jin, H. H. Liu, R. Gandhi, S. Kandula, R. Mahajan, M. Zhang, J. Rexford, and R. Wattenhofer. Dynamic scheduling of network updates. In *ACM SIGCOMM*, 2014.

[16] N. P. Katta, J. Rexford, and D. Walker. Incremental consistent updates. *HotSDN*, 2013.

[17] P. Kazemian, M. Chang, H. Zeng, G. Varghese, N. McKeown, and S. Whyte. Real time network policy checking using header space analysis. In *NSDI*, 2013.

[18] A. Khurshid, X. Zou, W. Zhou, M. Caesar, and P. B. Godfrey. VeriFlow: Verifying network-wide invariants in real time. In *NSDI*, 2013.

[19] H. H. Liu, X. Wu, M. Zhang, L. Yuan, R. Wattenhofer, and D. Maltz. zUpdate: Updating data center networks with zero loss. *ACM SIGCOMM*, 2013.

[20] A. Ludwig, M. Rost, D. Foucard, and S. Schmid. Good network updates for bad packets: Waypoint enforcement beyond destination-based routing policies. *HotNets*, 2014.

[21] R. Mahajan and R. Wattenhofer. On consistent updates in software defined networks. *HotNets*, 2013.

[22] J. McClurg, H. Hojjat, P. Cerny, and N. Foster. Efficient synthesis of network updates. *Programming Languages Design and Implementation*, 2015. to appear.

[23] A. Noyes, T. Warszawski, P. Černỳ, and N. Foster. Toward synthesis of network updates. *SYNT*, 2014.

[24] Open Network Foundation. OpenFlow switch specification v1.4, October 2013. https://www.opennetworking.org/.

[25] M. Reitblatt, N. Foster, J. Rexford, C. Schlesinger, and D. Walker. Abstractions for network update. In *ACM SIGCOMM*, 2012.

[26] R. Sherwood, G. Gibb, K.-K. Yap, G. Appenzeller, M. Casado, N. McKeown, and G. Parulkar. Can the production network be the testbed? *OSDI*, 2010.

[27] L. Shi, J. Fu, and X. Fu. Loop-free forwarding table updates with minimal link overflow. *International Conference on Communications*, 2009.

CoVisor: A Compositional Hypervisor for Software-Defined Networks

Xin Jin, Jennifer Gossels, Jennifer Rexford, David Walker
Princeton University

Abstract

We present CoVisor, a new kind of network hypervisor that enables, in a single network, the deployment of multiple control applications written in different programming languages and operating on different controller platforms. Unlike past hypervisors, which focused on *slicing* the network into disjoint parts for separate control by separate entities, CoVisor allows multiple controllers to *cooperate* on managing the same shared traffic. Consequently, network administrators can use CoVisor to assemble a collection of independently-developed "best of breed" applications—a firewall, a load balancer, a gateway, a router, a traffic monitor—and can apply those applications in combination, or separately, to the desired traffic. CoVisor also abstracts concrete topologies, providing custom virtual topologies in their place, and allows administrators to specify access controls that regulate the packets a given controller may see, modify, monitor, or reroute. The central technical contribution of the work is a new set of efficient algorithms for composing controller policies, for compiling virtual networks into concrete OpenFlow rules, and for efficiently processing controller rule updates. We have built a CoVisor prototype, and shown that it is several orders of magnitude faster than a naive implementation.

1 Introduction

A foundational principle of Software-Defined Networking (SDN) is to decouple control logic from vendor-specific hardware. Such a separation allows network administrators to deploy both the software and the hardware most suited to their needs, rather than being forced to compromise on one or both fronts because of the lack of availability of the perfect box. To fully realize this vision of freely assembling "best of breed" solutions, administrators should be able to run any combination of controller applications on their networks. If the optimal monitoring application is written in Python on Ryu [1] and the best routing application is written in Java on Floodlight [2], the administrator should be able to deploy both of them in the network.

A network hypervisor is a natural solution to this problem of bringing together disparate controllers. However, existing hypervisors [3, 4] restrict each controller to a distinct *slice* of network traffic. While useful in scenarios like multi-tenancy in which each tenant controls its own traffic, they do not enable multiple applications to collaboratively process the same traffic. Thus, an SDN hypervisor must be capable of more than just slicing. More specifically, in this paper, we show how to bring together the following key hypervisor features and implement them *efficiently* in a single, coherent system.

(1) Assembly of multiple controllers. A network administrator should be able to assemble multiple controllers in a flexible and configurable manner. Inspired by network programming languages like Frenetic [5], we compose data plane policies in three ways: *in parallel* (allow multiple controllers to act independently on the same packets at the same time), *sequentially* (allow one controller to process certain traffic before another), and *by overriding* (allow one controller to choose to act or to defer control to another controller). However, unlike Frenetic and related systems, our hypervisor is independent of the specific languages, libraries, or controller platforms used to construct client applications. Instead, the hypervisor intercepts and processes industry-standard OpenFlow messages, assembling and transforming them to match administrator-specified composition policies. Doing so efficiently requires new incremental algorithms for processing rule updates.

(2) Definition of abstract topologies. To protect the physical infrastructure, an administrator should be able to limit what each controller can see of the physical topology. Our hypervisor supports this by allowing the administrator to provide a custom virtual topology to each controller, thereby facilitating reuse of (physical) topology-independent code. For example, to a firewall controller the administrator may abstract the network as a "big virtual switch"; the firewall does not need to know the underlying topology to determine if a packet should be forwarded or dropped. In contrast, a routing controller needs the exact topology to perform its task effectively. In addition, topology abstraction helps the administrator implement complex functionality in a modular manner. Some switches, such as a gateway between an Ethernet island and the IP core, may play multiple roles in the network. The hypervisor can create one virtual switch for each role, assign each to a controller application precisely tailored to its single task, and compile policies written for the virtual network into the physical network.

(3) Protection against misbehaving controllers. In addition to restricting what a controller can see of the physical topology, an administrator may also want to impose fine-grained control on how a controller can process packets. This access control is important to protect against buggy or maliciously misbehaving third-party controllers. For example, a firewall controller should not be allowed to modify packets, and a MAC learner should not be able to inspect IP or TCP headers. The hypervisor enforces these restrictions by limiting the functionality of the virtual switches exposed to each controller.

The primary technical challenge surrounding the creation of such a fully featured hypervisor is efficiency. The hypervisor must host tens of controllers, each of which installs tens of thousands of rules. Complicating matters further, these controllers are updating rules constantly, as dictated by their application logic (e.g., traffic engineering, failure recovery, and attack detection [6, 7, 8, 9, 10, 11, 12, 13, 14]). The naive hypervisor design is to recompile the composed policy from scratch for every rule update, and then install in each switch's flow table the difference between the existing and updated policies. This strawman solution is prohibitively expensive in terms of both the time to compile the new policy and the time to install new rules on switches.

In this paper we present CoVisor, a hypervisor that exploits efficient new algorithms to compile and update policies from upstream controllers, each of which has its own view of the network topology. Figure 1 illustrates the CoVisor architecture. CoVisor serves as a transparent layer between controllers and the physical network. Each of the five applications shown at the top of Figure 1 is an unmodified SDN program running on its own controller; each controller outputs OpenFlow rules for the virtual topology shown below it, without any knowledge that this virtual topology does not physically exist. CoVisor intercepts the OpenFlow rules output by all five controllers and compiles them into a single policy for the physical network via a two-phase process.

First, CoVisor uses a novel algorithm to incrementally compose applications in the manner specified by the administrator. The key insight is that rule priorities form a convenient algebra to calculate priorities for new rules, obviating the need to recompile from scratch for every rule update. Second, CoVisor translates the composed policy into rules for the physical topology. Specifically, we develop a new compilation algorithm for the case of one physical switch mapped to multiple virtual switches. At both stages, CoVisor employs efficient data structures to further reduce compilation overhead by exploiting knowledge of the structure of policies provided by the access-control restrictions. After compiling the policy, CoVisor sends the necessary rule updates to switches.

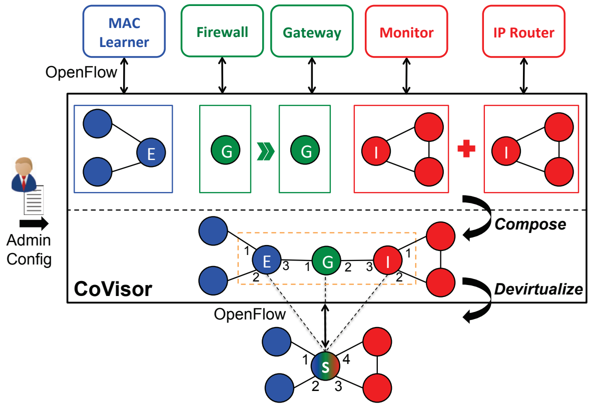


Figure 1: CoVisor overview.

At the far left of Figure 1, CoVisor takes configuration input from the administrator. These configuration responsibilities are threefold: (1) define how the policies of the controllers should be assembled; (2) create each controller's virtual network by specifying the components to be included and the physical-virtual mapping; and (3) state access control limitations for each controller.

In summary, we make the following contributions.

- We define the architecture of a new kind of compositional hypervisor, which allows applications written in different languages and on different controllers to process packets collaboratively.
- We develop a new algorithm to compile the parallel, sequential, and override operators introduced in earlier work [5, 15, 16] incrementally (§3).
- We develop a new, incremental algorithm to compile policies written for virtual topologies into rules for physical switches (§4).
- We employ customized data structures that leverage access-control restrictions, often a source of overhead, to further reduce compilation time (§5).

We describe our prototype in §6 and evaluation in §7. We have a brief discussion in §8, followed by related work in §9 and the conclusion in §10.

2 CoVisor Overview

This section provides an overview of CoVisor. CoVisor's features fall into two categories: (i) those that combine applications running on multiple controllers to produce a single flow table for each physical switch (§2.1); and (ii) those that limit an individual controller's view of the topology and packet-processing capabilities (§2.2).

To implement these features, CoVisor relies on a two-phase compilation process. The first phase assembles the policies of individual controllers, written for their own virtual networks, into a composed policy for the whole virtual network. The second phase compiles this composed policy for the virtual network into a policy for the physical network that realizes the intent expressed by the virtual policy. Algorithms for these phases are covered in §3 and §4, respectively.

2.1 Composition of Multiple Controllers

CoVisor allows network administrators to combine the packet-processing specifications of multiple controllers into a single specification for the physical network. We call these "packet-processing specifications" output by each controller *member policies* and the single specification a *composed policy*. In practice, the member policies are defined by OpenFlow commands issued from a controller to CoVisor. We use the terms *policy implementation* or just *implementation* to refer specifically to the list of OpenFlow rules used to express a policy.

The network administrator configures CoVisor to compose controllers with a simple language of commands. Let T range over policies defined in the command language. This language allows administrators to specify that some default action (a) should be applied to a set of packets, that a particular member policy (x) should be applied, that two separate policies should be applied in parallel ($T_1 + T_2$), that two separate policies should be applied in sequence ($T_1 \gg T_2$), or that one member policy should be applied, and if it fails to match a packet, some other policy should act as a default ($x \rhd T$). The following paragraphs explain these policies in greater detail.

Actions (a): The most basic composed policy is an atomic packet-processing action a. Such actions include any function from a packet to a set of packets implementable in OpenFlow, such as the actions to drop a packet ($drop$), to forward a packet out a particular port ($fwd(3)$), or to send a packet to the controller ($to\_controller(x)$).

Parallel operator ($+$): The parallel composition of two policies $T_1 + T_2$ operates by logically (though not necessarily physically) copying the packet, applying T_1 to one copy and T_2 to the other, and taking the union of the results. For example, let M be a monitoring policy and Q be a routing policy. If M counts packets based on source IP prefix and Q forwards packets based on destination IP prefix, $M + Q$ does both operations on all packets.

Sequential operator ($\gg$): The sequential operator enables two controllers to process traffic one after another. For example, let L be a load-balancing policy, and let Q be a routing policy. More specifically, for packets destined to anycast IP address 3.0.0.0, L rewrites the destination IP to a server replica's IP based on source IP prefix, and Q forwards packets based on destination IP prefix. To obtain the combined behavior of L and Q—to first rewrite the destination IP address and then forward the rewritten packet to the correct place—the network administrator uses the policy $L \gg Q$.

Command	Parameters
createVSw	pSw$_1$ <pSw$_2$, ..., pSw$_n$>
createVPort	vSw <pSw pPort>
createVLink	vSw$_1$ vPort$_1$ vSw$_2$ vPort$_2$
connectHost	vSw vPort host

Table 1: API to construct a virtual network. Brackets <> indicate optional arguments.

```
E  = createVSw S             // vswitch E
G  = createVSw S             // vswitch G
I  = createVSw S             // vswitch I
E1 = createVPort E S 1       // port 1 on E
E2 = createVPort E S 2       // port 2 on E
E3 = createVPort E           // port 3 on E
G1 = createVPort G           // port 1 on G
L1 = createVLink E 3 G 1     // link E-G
...        remaining commands omitted for brevity.
```

Figure 2: Administrator configuration to create (a subset of) the physical-virtual mapping shown in Figure 1.

Override operator ($\rhd$): Each controller x provides CoVisor with a member policy specifying how x wants the network to process packets. The policy $x \rhd T$ attempts to apply x's member policy to any incoming packet t. If x's policy does not specify how to handle t, then T is used as a default. For example, suppose one controller x is running an elephant flow routing application and another controller y is running an infrastructure routing application. If we want x to override y for elephant flow packets, y to route all regular traffic, and any packet not covered by either policy to be dropped, we use the policy $x \rhd (y \rhd drop)$.

2.2 Constraints on Individual Controllers

In addition to composing member policies, CoVisor allows the administrator to virtualize the underlying topology and restrict the packet-processing capabilities available to each controller. This helps administrators hide infrastructure information from third-party controllers, reuse topology-independent algorithms, and provide security against malicious or buggy control software.

2.2.1 Constraints on Topology Visibility

Rather than exposing the full details of the physical topology to each controller, CoVisor provides each with its own virtual topology. Table 1 shows the API to construct a custom virtual network. `createVSw` creates a virtual switch. It can be used to create two kinds of physical-virtual mappings as follows. (1) *many-to-one* (many physical switches map to one virtual switch): call the function once with a list of physical switch identifiers; (2) *one-to-many* (a single physical switch maps to many virtual switches): call the function multiple times with the same physical switch identifier.

`createVPort` creates a virtual port. To map it to a physical port, the administrator includes the corresponding physical switch and port number. `createVLink` creates a virtual link by connecting two virtual ports. `connectHost` connects a host to a virtual port.

Example. Consider the example physical-virtual topology mapping shown in Figure 1. The physical topology represents an enterprise network consisting of an Ethernet island (shown in blue in Figure 1) connected by a gateway router (multicolored and labeled *S*) to the IP core (red). We abstract gateway switch *S* to three virtual switches: E, G, and I. Figure 2 shows how the administrator uses CoVisor's API to create the virtual mapping.

These four commands allow the administrator to create one level of virtual topology on top of a physical network. To create multiple levels of topology abstraction, the administrator can run one CoVisor instance on top of another. Supporting this behavior in a single instance of CoVisor is part of our future work.

2.2.2 Constraints on Packet Handling

CoVisor imposes fine-grained access control on how a controller can process packets by virtualizing switch functionality. The administrator sets custom capabilities on each controller's virtual switches, thereby choosing which functionalities of the physical network to expose on a controller-by-controller basis.

Pattern: The administrator specifies which header fields a controller can match and how each field can be matched (i.e., exact-match, prefix-match, or arbitrary wildcard-match). CoVisor currently supports the 12 fields in the OpenFlow 1.0 specification, with prefix-match an option only for source and destination IP addresses.

Action: The administrator specifies the actions a controller can perform on matched packets. CoVisor currently supports the actions in the OpenFlow 1.0 specification, including forward, drop (indicated by an empty action list), and modify (the administrator determines which fields can be modified). The administrator also controls whether a controller can query packets and counters from switches and send packets to switches.

Example. In the example in Figure 1, the administrator can restrict the MAC learner to match only on source and destination MAC and inport and the firewall to match only on the five tuple. Also, the administrator can disallow both applications from modifying packets.

2.3 Handling Failures

Controllers, switches, and CoVisor itself can fail during operation. We describe how CoVisor responds to them.

Controller failure: The administrator configures CoVisor with a default policy for each controller to execute in the event of controller failure. The default policy is application-dependent. For example, a logical default for a firewall controller is *drop* (erase all installed rules and install a rule that drops all packets), because a firewall should fail safe. In contrast, the default policy for a monitoring controller can be *id* (identical, i.e., leave all rules in the switch), as monitoring rules are not critical to the operation of a network, and the counters can be reused if the monitoring controller recovers.

Switch failure: If a switch fails, all its rules are removed and CoVisor notifies the relevant controllers. Moreover, in the case of many-to-one virtualization, CoVisor allows the virtual switch to remain functional by rerouting traffic around the failed physical switch (if possible in the physical network).

Hypervisor failure: We currently do not deal with hypervisor failure. Replication techniques in distributed systems may be applied to CoVisor, but a full exploration is beyond the scope of this work.

3 Incremental Policy Compilation

Network management is a dynamic process. Applications update their policies in response to various network events, like a change in the traffic matrix, switch and link failures, and detection of attacks [6, 7, 8, 9, 10, 11, 12, 13, 14]. Therefore, CoVisor receives streams of member policy updates from controllers and has to recompile and update the composed policy frequently. In this section, we first review policy compilation and introduce a strawman solution, and then we describe an efficient solution based on a convenient algebra on rule priorities.

3.1 Background on Policy Compilation

The first stage of policy compilation entails combining member policies into a single composed policy. Controllers implement member policies by sending OpenFlow rules to CoVisor. A rule r is a triple $r = (p;m;a)$ where p is a priority, m is a match pattern, and a is an action list. Given a rule $r = (p;m;a)$, we use the notation $r.priority$ to refer to p, $r.match$ to refer to m, and $r.action$ to refer to a. We denote the set of packets matching $r.match$ as $r.mSet$. Now we describe how to compile each composition operator outlined in §2.1. We assume all policy implementations include only OpenFlow 1.0 rules and that each switch has a single flow table.

Parallel operator $(+)$**:** To compile $T_1 + T_2$, we first compile T_1 and T_2 into implementations R_1 and R_2. (In practice, each controller communicates its member policy to

Monitoring M_R
$(1;\ srcip = 1.0.0.0/24;\ count)$
$(0;\ *;\ drop)$

Routing Q_R
$(1;\ dstip = 2.0.0.1;\ fwd(1))$
$(1;\ dstip = 2.0.0.2;\ fwd(2))$
$(0;\ *;\ drop)$

Load balancing L_R
$(3;\ srcip = 0.0.0.0/2, dstip = 3.0.0.0;\ dstip = 2.0.0.1)$
$(1;\ dstip = 3.0.0.0;\ dstip = 2.0.0.2)$
$(0;\ *;\ drop)$

Elephant flow routing E_R
$(1;\ srcip = 1.0.0.0, dstip = 2.0.0.1;\ fwd(3))$

Parallel composition: $comp_{+}(M_R, Q_R)$
$(5;\ srcip = 1.0.0.0/24, dstip = 2.0.0.1;\ count, fwd(1))$
$(4;\ srcip = 1.0.0.0/24, dstip = 2.0.0.2;\ count, fwd(2))$
$(3;\ srcip = 1.0.0.0/24;\ count)$
$(2;\ dstip = 2.0.0.1;\ fwd(1))$
$(1;\ dstip = 2.0.0.2;\ fwd(2))$
$(0;\ *;\ drop)$

Sequential composition: $comp_{\gg}(L_R, Q_R)$
$(2;\ srcip = 0.0.0.0/2, dstip = 3.0.0.0;\ dstip = 2.0.0.1, fwd(1))$
$(1;\ dstip = 3.0.0.0;\ dstip = 2.0.0.2, fwd(2))$
$(0;\ *;\ drop)$

Override composition: $comp_{\triangleright}(E_R, Q_R)$
$(3;\ srcip = 1.0.0.0, dstip = 2.0.0.1;\ fwd(3))$
$(2;\ dstip = 2.0.0.1;\ fwd(1))$
$(1;\ dstip = 2.0.0.2;\ fwd(2))$
$(0;\ *;\ drop)$

Figure 3: Example of policy compilation.

CoVisor in an already compiled form. We explicitly include this step because it represents the base case of the recursive process.) Then, we compute $comp_{+}(R_1, R_2)$ by iterating over $(r_{1i}, r_{2j}) \in R_1 \times R_2$ where r_{1i} and r_{2j} are taken from R_1 and R_2, respectively, by priority in decreasing order. We produce a rule r in the composed implementation if the intersection of $r_{1i}.mSet$ and $r_{2j}.mSet$ is not empty. $r.match$ is the intersection of $r_{1i}.match$ and $r_{2j}.match$, and $r.actions$ is the union of $r_{1i}.actions$ and $r_{2j}.actions$. We defer priority assignment to later discussion in this subsection. Consider the example of $comp_{+}(M_R, Q_R)$ in Figure 3. Let $M_R = m_1, \ldots, m_n$ and $Q_R = q_1, \ldots, q_k$. We begin by considering m_1 and q_1. Since $m_1.mSet \cap q_1.mSet \neq \emptyset$, we produce a first rule r_1 in $comp_{+}(M_R, Q_R)$ with match pattern $\{srcip = 1.0.0.0/24, dstip = 2.0.0.1\}$ and action list $\{count, fwd(1)\}$. Composing all (m_i, q_j) pairs gives the composed policy implementation $comp_{+}(M_R, Q_R)$ of the policy composition $M + Q$.

Sequential operator ($\gg$): To compile $T_1 \gg T_2$, we again begin by generating implementations R_1 and R_2. Then, we compute $comp_{\gg}(R_1, R_2)$. As with $comp_{+}(R_1, R_2)$, we iterate over $(r_{1i}, r_{2j}) \in R_1 \times R_2$ where r_{1i} and r_{2j} are taken from R_1 and R_2, respectively, by priority in decreasing order. However, now we produce a rule r in the composed policy if the intersection of $r_{2j}.mSet$ and the set of packets produced by applying $r_{1i}.action$ to all packets in $r_{1i}.mSet$ is not empty. Consider the example of $comp_{\gg}(L_R, Q_R)$ in Figure 3. Again, we begin iterating over $(l_i, q_j) \in L_R \times Q_R$ pairs by considering l_1 and q_1. Applying $l_1.action$ to all packets in $l_1.mSet$ gives the set of packets matching pattern $\{srcip = 0.0.0.0/2, dstip = 2.0.0.1\}$. The intersection of this set and $q_1.mSet$ is not empty. Hence, we generate the first rule in the composed policy implementation with match pattern $\{srcip = 0.0.0.0/2, dstip = 3.0.0.0\}$ and action list $\{dstip = 2.0.0.1, fwd(1)\}$. Repeating this process for all (l_i, q_j) pairs yields $comp_{\gg}(L_R, Q_R)$, the implementation of $L \gg Q$.

Routing Q_R
$(1;\ dstip = 2.0.0.1;\ fwd(1))$
$(1;\ dstip = 2.0.0.2;\ fwd(2))$
(**1**; **dstip=2.0.0.3**; **fwd(3)**)
$(0;\ *;\ drop)$

Parallel composition: $comp_{+}(M_R, Q_R)$
(**7**; **srcip=1.0.0.0/24,dstip=2.0.0.1**; **fwd(1),count**)
(**6**; **srcip=1.0.0.0/24,dstip=2.0.0.2**; **fwd(2),count**)
(**5**; **srcip=1.0.0.0/24,dstip=2.0.0.3**; **fwd(3),count**)
(**4**; **srcip=1.0.0.0/24**; **count**)
(**3**; **dstip=2.0.0.1**; **fwd(1)**)
(**2**; **dstip=2.0.0.2**; **fwd(2)**)
(**1**; **dstip=2.0.0.3**; **fwd(3)**)
$(0;\ *;\ drop)$

Figure 4: Example of updating policy composition. Strawman solution.

Override operator ($\triangleright$): To compile $T_1 \triangleright T_2$, we again begin by generating implementations R_1 and R_2. Then, we compute $comp_{\triangleright}(R_1, R_2)$ by stacking R_1 on top of R_2 with higher priority. For example in Figure 3, to compile $comp_{\triangleright}(E_R, Q_R)$, we put E_R's rules above Q_R's rules. Thus, packets with source IP 1.0.0.0 and destination IP 2.0.0.1 will be forwarded to port 3, and other packets with destination IP 2.0.0.1 will be forwarded to port 1.

Priority assignment and policy update problem: Recall that a rule r is a triple $(r.priority; r.match; r.action)$. Thus far, we have explained how to generate a list of $(match; action)$ pairs, or *pseudo-rules*. Our list of

Monitoring M_R
$(1;\ srcip = 1.0.0.0/24;\ count)$
$(0;\ *;\ drop)$

Routing Q_R
$(1;\ dstip = 2.0.0.1;\ fwd(1))$
$(1;\ dstip = 2.0.0.2;\ fwd(2))$
(1; dstip=2.0.0.3; fwd(3))
$(0;\ *;\ drop)$

Load balancing L_R
$(3;\ srcip = 0.0.0.0/2, dstip = 3.0.0.0;\ dstip = 2.0.0.1)$
(2; srcip=0.0.0.0/1,dstip=3.0.0.0; dstip=2.0.0.3)
$(1;\ dstip = 3.0.0.0;\ dstip = 2.0.0.2)$
$(0;\ *;\ drop)$

Elephant flow routing E_R
$(1;\ srcip = 1.0.0.0, dstip = 2.0.0.1;\ fwd(3))$

Parallel composition: $comp_{+}(M_R, Q_R)$
$(2;\ srcip = 1.0.0.0/24, dstip = 2.0.0.1;\ fwd(1), count)$
$(2;\ srcip = 1.0.0.0/24, dstip = 2.0.0.2;\ fwd(2), count)$
(2; srcip=1.0.0.0/24,dstip=2.0.0.3; fwd(3),count)
$(1;\ srcip = 1.0.0.0/24;\ count)$
$(1;\ dstip = 2.0.0.1;\ fwd(1))$
$(1;\ dstip = 2.0.0.2;\ fwd(2))$
(1; dstip=2.0.0.3; fwd(3))
$(0;\ *;\ drop)$

Sequential composition: $comp_{\gg}(L_R, Q_R)$
$(25;\ srcip = 0.0.0.0/2, dstip = 3.0.0.0;\ dstip = 2.0.0.1, fwd(1))$
(17; srcip=0.0.0.0/1,dstip=3.0.0.0; dstip=2.0.0.3,fwd(3))
$(9;\ dstip = 3.0.0.0;\ dstip = 2.0.0.2, fwd(2))$
$(0;\ *;\ drop)$

Override composition: $comp_{\triangleright}(E_R, Q_R)$
$(9;\ srcip = 1.0.0.0, dstip = 2.0.0.1;\ fwd(3))$
$(1;\ dstip = 2.0.0.1;\ fwd(1))$
$(1;\ dstip = 2.0.0.2;\ fwd(2))$
(1; dstip=2.0.0.3; fwd(3))
$(0;\ *;\ drop)$

Figure 5: Example of incremental update.

pseudo-rules is prioritized in the sense that each pseudo-rule's position indicates its relative priority, but we have not addressed how to assign a particular priority value to each pseudo-rule. Priority assignment is important for minimizing the overhead of policy update. Ideally, a single rule addition in one member policy implementation should not require recomputing the entire composed policy from scratch, nor should it require clearing the physical switch's flow table and installing thousands of flowmods. (A flowmod is an OpenFlow message to update a rule in a switch.) In concrete terms, the update problem involves minimizing the following two overheads:

- **Computation overhead:** The number of rule pairs over which the composition function *comp* iterates to recompile the composed policy.
- **Rule update overhead:** The number of flowmods needed to update a switch to the new policy.

Strawman solution: The strawman solution is to assign priorities to rules in the composed implementation from bottom to top starting from 0 by increment of 1. Then, it installs the difference between the old implementation and the new one. For example, the priorities of rules in Figure 3 are assigned in this way. This approach incurs high computation and rule update overhead, because it requires recompiling the whole policy to determine each rule's new relative position and updates rules that only change priorities. For example, when a new rule is inserted to Q_R (in bold in Figure 4), although only the third and the seventh rules in $comp_{+}(M_R, Q_R)$ are new, five rules change their priorities. We have to update these five existing rules as well as add two new rules. Rules in bold in Figure 4 count toward this rule update overhead.

3.2 Incremental Update

Ideally, the priority of rule r in the composed implementation is a function solely of the rules in the member implementations from which it is generated. In this way, any updates of other rules in member implementations will not affect r. We observe that rule priorities form a convenient algebra which allows us to achieve this goal.

Add for parallel composition: Let R be the composed implementation of $comp_{+}(R_1, R_2)$. If rule $r_k \in R$ is composed from $r_{1i} \in R_1$ and $r_{2j} \in R_2$, then $r_k.priority$ is the sum of $r_{1i}.priority$ and $r_{2j}.priority$:

$$r_k.priority = r_{1i}.priority + r_{2j}.priority. \qquad (1)$$

We show the example of $comp_{+}(M_R, Q_R)$ in Figure 5. The first rule in $comp_{+}(M_R, Q_R)$ is composed from m_1 and q_1. Hence, its priority is $m_1.priority + q_1.priority = 2$. Suppose a new rule (in bold in Figure 5) is inserted to Q_R. We only need to iterate over rule pairs (m_i, q_3) for all $m_i \in M_R$, rather than iterate over all the rule pairs. This generates two new rules (in bold in Figure 5). All existing rules do not change.

Concatenate for sequential composition: Let R be the composed implementation of $comp_{\gg}(R_1, R_2)$. If $r_k \in R$ is composed from $r_{1i} \in R_1$ and $r_{2j} \in R_2$, then $r_k.priority$ is the concatenation of $r_{1i}.priority$ and $r_{2j}.priority$:

$$r_k.priority = r_{1i}.priority \circ r_{2j}.priority. \qquad (2)$$

Symbol $\circ$ in Equation 2 represents the concatenation of two priorities, where each priority is represented as a fixed-width bit string. Concatenation enforces a *lexicographic ordering* on the pair of priorities. Specifically, let $a_1 = b_1 \circ c_1$ and $a_2 = b_2 \circ c_2$. Then $a_1 > a_2$ if and only if $\big(b_1 > b_2$ or $(b_1 = b_2$ and $c_1 > c_2)\big)$, and $a_1 = a_2$ if and only if $(b_1 = b_2$ and $c_1 = c_2)$. In practice, concatenation is computed as follows. Let r_{2j} be in the range $[0, MAX_{R_2})$ where $MAX_{R_2} - 1$ is the highest priority that R_2 may use[1]. Then $r_k.priority$ is computed by

$$r_k.priority = r_{1i}.priority \times MAX_{R_2} + r_{2j}.priority.$$

We show the example of $comp_{\gg}(L_R, Q_R)$ in Figure 5. Let $MAX_{Q_R} = 8$. The first rule in $comp_{\gg}(L_R, Q_R)$ is composed from l_1 and q_1. Thus, its priority is $l_1.priority \times 8 + q_1.priority = 25$. Suppose a new rule is inserted to L_R $\big($in bold in Figure 5$\big)$. We only need to iterate over rule pairs (l_3, q_j) for all $q_j \in Q_R$. This generates a new rule with priority 17 $\big($in bold in Figure 5$\big)$. All existing rules do not change.

Stack for override composition: Let R be the composed implementation of $comp_{\triangleright}(R_1, R_2)$, and let R_2's priority space be $[0, MAX_{R_2})$. To assign priorities in R, we increase the priorities of R_1's rules by MAX_{R_2} and keep the priorities of R_2's rules unchanged. This process essentially stacks R_1's priority space on top of R_2's priority space. Specifically, let $r_k \in R$. By definition of $comp_{\triangleright}$, r_k is in either R_1 or R_2. Let $r_k.mPriority$ be r_k's priority in the member implementation from which it comes. We assign priority to r_k as follows.

$$r_k.priority = \begin{cases} r_k.mPriority + MAX_{R_2} & \text{if } r_k \in R_1 \\ r_k.mPriority, & \text{if } r_k \in R_2 \end{cases} \quad (3)$$

We show the example of $E_R \triangleright Q_R$ in Figure 5. Let $MAX_{Q_R} = 8$. The first rule in $comp_{\triangleright}(E_R, Q_R)$ is generated from e_1, so it is assigned priority $e_1.priority + 8 = 9$. The second rule in $comp_{\triangleright}(E_R, Q_R)$ is generated from q_1, so it is assigned priority $q_1.priority = 1$. When a new rule q_3 that matches $dstip = 1.0.0.3$ is inserted to Q_R, we simply add a new rule with priority 1 (in bold in Figure 5) to $comp_{\triangleright}(E_R, Q_R)$ without affecting existing rules.

Remark 1 *We show the proofs of correctness for parallel and sequential composition in [17]. A similar proof for override composition is straightforward.*

With the algebra on rule priorities above, CoVisor processes the three kinds of rule updates as follows. Let R be the composed policy implementation of R_1 and R_2.

Rule addition: When a new rule r_1^* is added to R_1 (or r_2^* to R_2), CoVisor composes this rule with each rule in R_2 (or R_1). It assigns priorities to new rules according to Equations 1, 2, and 3 and installs them to switches. All existing rules are untouched.

Algorithm 1 Symbolic path generation

```
function GENPATHS(pkt)
    pkt.children ← {evaluate policy on pkt}
    for all child in pkt.children do
        if not child.reachedEgress() then
            GENPATHS(child)
```

Rule deletion: When an old rule r_1^* is deleted from R_1 (or r_2^* from R_2), CoVisor finds all rules in R that are composed from this rule and deletes them from switches. All other rules are untouched.

Rule modification: Modifying a rule is equivalent to deleting an old rule and then inserting a new rule.

4 Compiling Topology Transformations

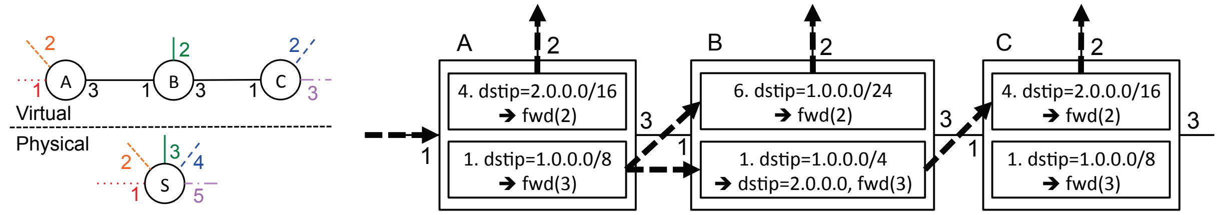
The first phase of compilation (§3) generates a composed policy for the virtual network. The second phase, which we describe in this section, compiles a policy for the virtual topology into one for the physical network. It comprises two sub-cases as described in §2.2.1: many-to-one and one-to-many. One-to-one is a degenerate case of these two. While previous work has explored compilation of the many-to-one case [18, 19], there does not exist any compilation algorithm for the one-to-many case. Pyretic [20] offers the one-to-many feature but implements it by sending the first packet of each flow to the controller and then installing micro-flow rules, a strategy which incurs prohibitive overhead. We present the first compilation algorithm for the one-to-many case.

Our algorithm is a novel combination of symbolic analysis [21] and incremental sequential composition. Intuitively, we inject a symbolic packet into the virtual network, follow all possible paths to egress ports, and sequentially compose the rules along each path. In this way, we derive rules for the physical switch to process traffic as intended by the controller's policy for the virtual network. To handle rule updates incrementally, we keep all the symbolic paths computed during this analysis and minimally modify them when the virtual policy changes. We divide our description into three parts: symbolic path generation (§4.1), sequential composition on symbolic paths (§4.2), and incremental update (§4.3).

4.1 Symbolic Path Generation

For each ingress port of the virtual network, we inject a single symbolic packet with wildcards in all fields (except *inport*). At every hop, we evaluate the policy on the packet, which generates zero, one, or more symbolic

[1] CoVisor limits the priority space of each member policy, because the bits for priority in hardware are limited in practice.



(a) Topology. Colors and line types indicate physical-virtual mapping.

(b) Flow tables of virtual switches.

Figure 6: One-to-many virtualization.

packets. We follow the generated symbolic packets until they all reach egress ports. Together, these symbolic paths form a tree rooted at the ingress port.

Algorithm 1 shows pseudocode for the path generation algorithm. In Line 2, we create all child packets that can result from evaluating the policy on `pkt`—one child packet for each rule r that `pkt` matches. As we construct the tree, we update `child`'s header, which denotes the subset of traffic represented by the symbolic packet, according to the information encoded in the rule responsible for generating `child` from `pkt`. By doing so, we avoid creating branches for paths that no packet could possibly follow.

We use the example in Figure 6 to illustrate the process. Figure 6(a) shows a physical-virtual topology mapping in which a physical switch S is virtualized to three virtual switches, A, B and C. The mapping between physical and virtual ports is color- and line type-coded. Figure 6(b) shows the policy of each virtual switch.

We inject a symbolic packet with header $*$, denoting wildcards in all fields, into port 1 of A. When we apply A's policy to this packet, we generate two child symbolic packets, p_1 and p_2. p_1 has destination IP 2.0.0.0/16, matches the first rule in A's policy, A_{R1}, and leaves the network at port 2 of A; p_2 has destination IP 1.0.0.0/8, matches A's second rule, A_{R2}, and reaches port 1 of B. We then evaluate B's policy on p_2, again generating two symbolic packets, p_{21} and p_{22}. p_{21} matches B_{R1} and leaves the network at port 2 of B; p_{22} matches B_{R2}, enters C at port 1, matches C_{R1}, and finally leaves the network at port 2 of C. In total, we get the following three symbolic paths: (1) $p_1 : A_{R1}$; (2) $p_{21} : A_{R2} \rightarrow B_{R1}$; and (3) $p_{22} : A_{R2} \rightarrow B_{R2} \rightarrow C_{R1}$.

4.2 Sequential Composition

For each symbolic path, we sequentially compose all the rules along its edges to generate a single rule. Then, we derive a final rule for the physical switch by adding a match on the *inport* value of the symbolic packet at the root of the tree. Returning to our example in Figure 6, the first symbolic path contains only A_{R1}. By adding port 1 to its match, we get the first rule for physical switch S.

$$S_{R1} = \big(4;\ inport = 1, dstip = 2.0.0.0/16;\ fwd(2)\big).$$

Adding $inport = 1$ is necessary because traffic that enters port 3 of C (port 5 of S) with destination IP 2.0.0.0/16 will be forwarded to port 2 of C (port 4 of S). Similarly, for the second and third symbolic paths, we evaluate $comp_{\gg}(A_{R2}, B_{R1})$ and $comp_{\gg}\big(comp_{\gg}(A_{R2}, B_{R2}), C_{R1}\big)$, respectively. We assume the priority space for each switch is $[0, 8)$. After adding ingress port, we obtain two more rules.

$$S_{R2} = \big(14; inport = 1, dstip = 1.0.0.0/24; fwd(3)\big)$$
$$S_{R3} = \big(76; inport = 1, dstip = 1.0.0.0/8;$$
$$dstip = 2.0.0.0, fwd(4)\big)$$

Priority assignment: Because symbolic paths may have different lengths, for the devirtualization phase of compilation we need to augment the priority assignment algorithm for sequential composition presented in §3.2. For example, from sequential composition we get priorities 4, 14, and 76 for rules S_{R1}, S_{R2}, and S_{R3}, respectively. But, with these priorities, traffic entering port 1 at S with source IP 1.0.0.0/24 would match S_{R3} rather than S_{R2}, even though S_{R2} should have a higher priority than S_{R3}. This mismatch happens because S_{R2} is calculated from a path with only two hops (its priority is $1 \circ 6 = 14$) and S_{R3} is calculated from one with three hops ($1 \circ 1 \circ 4 = 76$). To address the mismatch, we set a hop length l^*. If a path is fewer than l^* hops, we pad 0s to the concatenation of the rule priorities. In practice, we use the number of switches in the virtual topology as l^*, as a path will have more than that number of hops only if the virtual policy contains a loop. This modified algorithm correctly orders S_{R2} and S_{R3}, assigning them respective priorities of $1 \circ 6 \circ 0 = 112$ and $4 \circ 0 \circ 0 = 256$. Figure 7 shows the rules for S with priorities calculated in this manner. We repeat the above procedure for all ingress ports of the virtual topology to get the final policy for S.

4.3 Incremental Update

By storing all the symbolic paths we generate when compiling a policy and partially modifying them upon a rule insertion or deletion, we can incrementally update a policy. This strategy obviates the need to compile the whole

Flow table of S
$(256;\ inport = 1, dstip = 2.0.0.0/16;\ fwd(2))$
$(112;\ inport = 1, dstip = 1.0.0.0/24;\ fwd(3))$
$(76;\ inport = 1, dstip = 1.0.0.0/8;\ dstip = 2.0.0.0, fwd(4))$

Figure 7: Flow table of switch S in Figure 6.

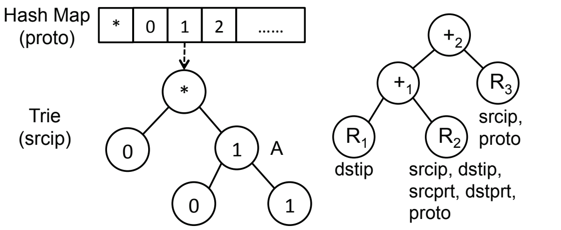
policy from scratch upon every rule update. In particular, when virtual switch V receives a rule update, we reevaluate V's policy on all symbolic packets that enter V. As a result, we may generate new symbolic packets, which we then follow until they reach egress ports. V's policy update may also modify the headers of or eliminate existing symbolic packets. Accordingly, we update the paths of modified symbolic packets and remove the paths of deleted packets. Then, we add and remove rules from the physical switch as described in §4.2. Our priority assignment algorithm ensures that these rule additions and deletions do not affect existing rules generated from symbolic paths that have not changed.

5 Exploiting Policy Structures

CoVisor imposes fine-grained access control on how each controller can match and modify packets. These restrictions both enhance security and provide hints that allow CoVisor to further optimize the compilation process. First, by knowing which fields individual policies match on and modify, we can build custom data structures to index rules, instead of resorting to general R-tree-based data structures for multi-dimensional classifiers as in [22, 23, 24]. Second, by correlating the matched or modified fields of two policies being composed, we can simplify their indexing data structures by only considering the fields they *both* care about.

We first describe the optimization problem, and then we show how to use the above two insights to solve it. For ease of explanation, we first assume that member policies are connected by the parallel operator. Later, we'll describe how to handle the sequential and override operators. Now suppose we have a parallel composition $T_1 + T_2$ with implementation $comp_+(R_1, R_2)$, and a new rule, r_1^*, is inserted into R_1. With our incremental update algorithm (§3.2), we need to iterate over all (r_1^*, r_{2j}) pairs where $r_{2j} \in R_2$. The iteration processes $|R_2|$ pairs in total, where $|R_2|$ denotes the number of rules in R_2. However, if we know the structure of R_2, we can index its rules in a way that allows us to skip the rules that don't intersect with r_1^*, thereby further reducing computation overhead.

Index policies based on structure hints: Our goal is to reduce the number of rule pairs to iterate in compilation. A policy's structure indicates which fields should be indexed and how. For example, if R_2 is permitted only to do exact-match on destination MAC, then we can store its rules in a hash map keyed on destination MAC. If r_1^* also does exact-match on destination MAC, we simply use the destination MAC as key to search for rules in R_2's hash map. No rules in R_2 besides those stored under this key can intersect with r_1^*, because they differ on destination MAC. If r_1^* wildcards destination MAC, we return all rules in R_2, as they all intersect with r_1^*.



(a) Example rule index.

(b) Example syntax tree.

Figure 8: Example of exploiting policy structures.

The preceding example is a simple case in which R_2 matches on one field. In general, a policy may match on multiple fields. We use single-field indexes (hash table for exact-match, trie for prefix-match, list for arbitrary wildcard-match) as building blocks to build a *multi-layer index* for multiple fields. Specifically, we first choose one field f_1 the policy can match and index the policy on this field. We store all rules with the same value in f_1 in the same bucket of the index. This forms the first layer of the index. Then we choose the second field f_2 and index rules in each f_1 bucket on f_2. We repeat this process for all the fields on which the policy can match. We choose the order of fields according to simple heuristics like preferring exact-match fields to prefix-match fields. In practice, a policy normally matches on a small number of fields, which means the number of layers is small.

Consider a policy that does exact-match on $proto$ (protocol number) and prefix-match on $srcip$. We first index the policy based on $proto$. All rules with the same value in $proto$ go to the same bucket, as shown in Figure 8(a). Note that the hash map contains a bucket keyed on $*$ for rules that do not match on $proto$. Then, we index all the rules that contain the same $proto$ value on $srcip$. Because our example policy does prefix match on $srcip$, the second level of our multi-layer index comprises a trie for each bucket in the hash map. Figure 8(a) shows this second level for rules with $proto = 1$; bucket A contains all the rules with $proto = 1$ and $srcip = 128.0.0.0/1$.

Correlate policy structures to reduce indexing fields: When composing policies, we can leverage the information we know about both to reduce the work we do to index each. Suppose R_1 matches on $dstip$ and R_2 matches on the five tuple ($srcip$, $dstip$, $srcport$, $dstport$, $proto$). Instead of storing R_2 in a five-layer index, we need only index the $dstip$. Because $dstip$ is the only field on which any rule r_1^* added to R_1 can match, r_1^* will intersect with a rule in R_2 as long as they intersect on $dstip$. Formally, let $R_i.fields$ be the set of fields on which R_i matches and $R_i.index$ be the set of fields R_i indexes. Given R_i and R_j in a composition, we have

$$R_i.index = R_j.index = R_i.fields \cap R_j.fields. \tag{4}$$

Back to our example, we have $R_1.index = R_2.index = R_1.fields \cap R_2.fields = \{dstip\}$.

A policy R_i itself may be composed from other policies R_j and R_k. Unlike in the previous example, we do not *a priori* know $R_i.fields$ and instead rely on the observation that a rule in a composed policy can match on a field f if and only if at least one of its component member policies can match on f. Hence, we get

$$R_i.fields = R_j.fields \cup R_k.fields. \tag{5}$$

Let's look at an example $(R_1 + R_2) + R_3$, which we show as a syntax tree in Figure 8(b). Initially, we know the match fields only for the leaf nodes. Then we calculate the match fields for node $+_1$ with $R_1.fields \cup R_2.fields = \{srcip, dstip, srcprt, dstprt, proto\}$. Then, we use Equation (4) to index $+_1$ and R_3 with $+_1.fields \cap R_3.fields = \{srcip, proto\}$.

Sequential and override composition: Suppose we have sequential composition $T_1 \gg T_2$ with implementation $comp_{\gg}(R_1, R_2)$. Then $R_1.fields$ not only contains the fields R_1 matches but also the fields it modifies in its action set. This is because, for $r_1 \in R_1$ and $r_2 \in R_2$, the pair (r_1, r_2) generates a rule for the composed policy if the intersection of $r_2.mSet$ and the set of packets resulting from applying $r_1.action$ to $r_1.mSet$ is not empty. Similarly, when we index R_1, the key for any rule r_{1i} is the value resulting from applying $r_1.action$ to $r_1.match$. We do not need to index policies for override composition, since we directly stack their rules.

6 Implementation

We implemented a prototype of CoVisor with 4000+ lines of Java code added to and modifying OpenVirteX [3]. We replaced the core logic of OpenVirteX, which isolates multiple controllers, with our composition and incremental update logic (§3). To OpenVirteX's built-in many-to-one virtualization, we added support for the one-to-many abstraction and our proactive compilation algorithm (§4). We further optimized compilation by exploiting the structure of policies as described in §5. We used `HashMap` in the Java standard library [25] to index rules with exact-match fields and `RadixTree` in the Concurrent-trees library [26] to index rules with prefix-match fields. Given a key (e.g., 1.0.0.0/16), `RadixTree` in the Concurrent-trees library only returns values for keys starting with this key (e.g., 1.0.0.0/24 and 1.0.0.0/30). We modified it to also give values for keys included by this key (e.g., 1.0.0.0/8). CoVisor currently supports the OpenFlow flowmod message; other commands, such as barrier messages and querying counters, will be supported in later versions.

7 Evaluation

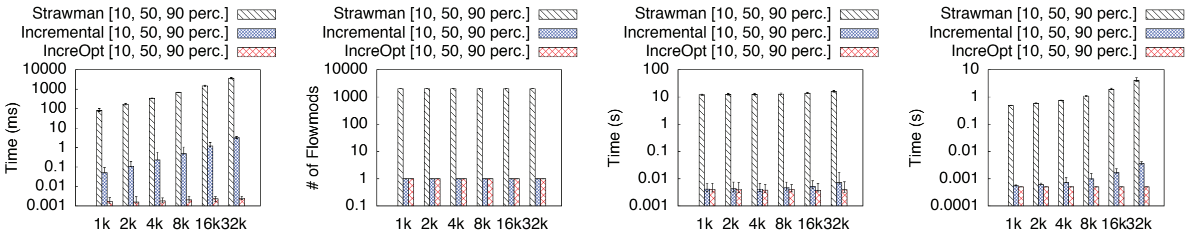
7.1 Methodology

Experiment Setup: We evaluate CoVisor under three scenarios, the first two of which evaluate composition efficiency and the third of which evaluates devirtualization efficiency. In each scenario, we stress CoVisor with a wide range of policy sizes. Since compiling policies to individual physical switches is independent in these scenarios, we show the results for a single physical switch. We run CoVisor on Mininet [27] and use Floodlight controllers [2]. The server is equipped with an Intel XEON W5580 processor with 8 cores and 6GB RAM. We describe each scenario in more detail below.

- **L2 Monitor + L2 Router:** L2 Monitor counts packets for source MAC and destination MAC pairs; L2 Router forwards packets based on destination MAC. The MAC addresses are randomly generated.
- **L3-L4 Firewall ≫ L3 Router:** L3-L4 Firewall filters packets based on the five tuple; L3 Router forwards packets according to destination IP prefix. The firewall policy is generated from ClassBench [28], a tool for benchmarking firewalls. The L3 router policy is generated with IP prefixes extracted from the firewall policy.
- **Gateway virtualization:** This is the topology virtualization discussed in §2.2.1. A switch that connects an Ethernet island to the IP core is abstracted to three virtual switches, which operate as a MAC learner, gateway, and IP router.

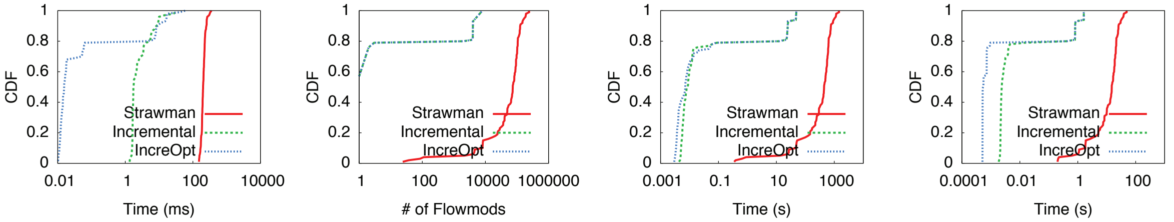
Metrics: We use the following metrics to measure efficiency. The thick bars in Figures 9 and 11 indicate the median, and the error bars show the 10th and 90th percentiles.

- **Compilation time:** The time to compile the policy composition or topology devirtualization.
- **Rule update overhead:** The number of flowmods to update the switch to the new flow table.
- **Total update time:** The sum of compilation time, rule update time, and additional system overhead like OpenFlow message (un)marshalling. Since hardware switches and software switches takes very different time in rule updates, we show both of them. As the software switches in Mininet do not mimic the rule update latency of hardware switches and do not give accurate timing on the actual rule installation in software switches, we use the rule update latency in [29] for hardware switches and that



(a) Compilation Time (b) Rule Update Overhead (c) Total Update Time (Hardware) (d) Total Update Time (Software)

Figure 9: Per-rule update overhead of L2 Monitor + L2 Router as a function of L2 Router size (log-log scale).



(a) Compilation Time (b) Rule Update Overhead (c) Total Update Time (Hardware) (d) Total Update Time (Software)

Figure 10: Per-rule update overhead of L3-L4 Firewall ≫ L3 Router (x-axis log scale).

in [30] for software switches when calculating rule update times.

Comparison: We compare the following approaches.

- **Strawman:** Recompile the new policy from scratch for every policy update.
- **Incremental:** Incrementally compile the new policy using our algebra of rule priorities for policy composition (§3) and keeping symbolic path information for topology devirtualization (§4).
- **IncreOpt:** Further optimize Incremental by exploiting the structures of policies (§5).

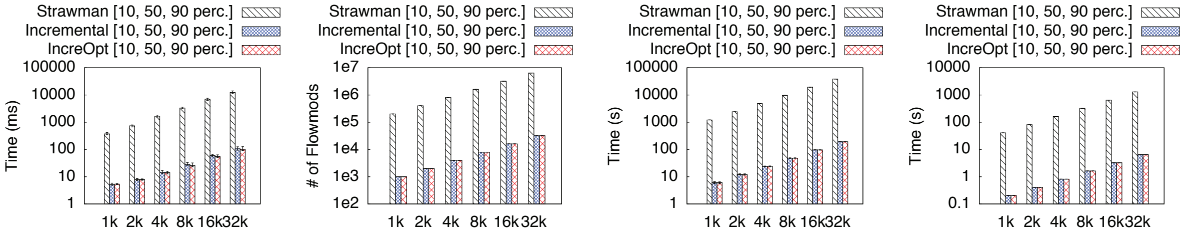
7.2 Composition Efficiency

Figure 9 shows the result of L2 Monitor + L2 Router. In this experiment, we initialize the L2 Monitor policy with 1000 rules, and then add 10 rules to measure the overhead for each. We repeat this process 10 times. We vary the size N of L2 Router policy from 1000 to 32,000 to show how overhead increases with larger policies. Figure 9(a) shows the compilation time. As expected, the compilation time of Strawman and Incremental increases with the policy size, because larger policies force our algorithm to consider more rule pairs. Since Strawman recompiles the whole policy, it is by far the slowest. On the other hand, IncreOpt has almost constant compilation time, because it indexes L2 Router's rules in a hash table keyed on destination MAC. When a rule is inserted to L2 Monitor's policy, the algorithm simply uses the rule's destination MAC to look up rules in the hash table.

Figure 9(b) shows the rule update overhead in terms of number of rules (same for hardware and software switches). Because of its naive priority assignment scheme, Strawman unnecessarily changes priorities of many existing rules and thus generates more flowmods than Incremental and IncreOpt. Incremental and IncreOpt generate the same policy, and therefore they have the same rule update overhead. We also observe that the rule update overhead does not increase with the size of L2 Router's policy. This is because the size of L2 Monitor's policy is fixed, and each monitor rule only intersects with one rule in L2 Router, since they both do exact-match on destination MAC.

Finally, Figures 9(c) and 9(d) show the total time. Notably, Incremental and IncreOpt are significantly faster than Strawman, and the gap between Incremental and IncreOpt is larger when using software switches. This is because software switches update rules faster than hardware switches, and therefore the compilation time accounts for a larger fraction of the total time for software switches.

Figure 10 shows the result of L3-L4 Firewall ≫ L3 Router. As before, we initialize L3-L4 Firewall's policy with 1000 rules and add 10 rules. Since the trend is similar to Figure 9 when we vary the size N of L3 Router, we instead show the CDF when L3 Router policy has 8,000 rules. Figure 10(a) shows the compilation time. Again, Strawman is several orders of magnitude slower than Incremental and IncreOpt. However, unlike in our previous experiment, we see a stepwise behavior of Incremental, and the difference between Incremental and IncreOpt also disappears after 80th percentile. This is an artifact of the content of L3-L4 Firewall from ClassBench. The firewall policy comprises approximately 80% rules matching on very specific destination IP prefix (/31, /32) and around 20% rules matching very general destination IP prefix



(a) Compilation Time (b) Rule Update Overhead (c) Total Update Time (Hardware) (d) Total Update Time (Software)

Figure 11: The switch connecting an Ethernet island to the IP core is virtualized to switches that operate as MAC learner, gateway, and IP router. Figures show the overhead of adding a host to the Ethernet island as a function of IP router policy size (log-log scale).

(/1, /0). A firewall rule with a very specific destination IP prefix only composes with a few router rules, in which case IncreOpt processes fewer rule pairs in compilation than Incremental. On the other hand, a firewall rule with a very general destination IP prefix like /1 or /0 composes with half or all rules in the router policy, in which case Incremental and IncreOpt process a similar number of rule pairs and have similar compilation time. This reasoning also explains the shape of Incremental and IncreOpt in Figures 10(b), 10(c) and 10(d). Finally, note that the overhead of inserting a new rule to L3-L4 Firewall by Incremental and IncreOpt is bounded by the number of rules in L3 Router, while that by Strawman is bounded by the product of the number of rules in L3-L4 Firewall and L3 Router.

7.3 Devirtualization Efficiency

We use the gateway scenario to evaluate the efficiency of the devirtualization phase of compilation. In this experiment, we have 100 hosts in the Ethernet island. The MAC learner installs forwarding rules for connections between host pairs. To the Ethernet island, switch *G* simply appears as another host; hosts use *G*'s MAC as destination MAC when they want to reach hosts across the IP core. We initialize the MAC learner policy with 1000 rules in switch *E*. Then, we add a new host to the Ethernet island. When the new host tries to talk to another host across the IP core, the MAC learner adds two rules to establish a bidirectional connection between the host and switch *G*. To compile this update, we compose the two new rules with the existing rules in switches *G* and *I*. The gateway policy at *G* is simply a MAC-rewriting repeater and ARP server. The IP router forwards packets based on destination IP prefix. We vary the size of the IP router policy at *I* from 1000 to 32,000 to evaluate how the overhead increases with larger policies.

Figure 11 shows the overhead. Strawman exhibits a long compilation time, as it has to recompile the policy from scratch. Strawman also generates more flowmods than necessary, because its priority assignment scheme may change the priorities of existing rules. In contrast, Incremental and IncreOpt incur significantly less overhead, because they keep all the symbolic paths and only need to change a few upon receiving the new rules. Finally, we notice that Incremental and IncreOpt do not show much difference in this experiment and the absolute values of total update time are high. This is because the MAC learner policy in switch *E* and the IP router policy in switch *I* match on different fields. Thus, when we do sequential composition on virtual paths, Incremental and IncreOpt iterate over a similar number of rule pairs and the result policy is almost a cross-product of the two policies at *E* and *I*. The cross-product is inevitable when compiling to a single flow table as the two policies match on different fields. Finally, we note that the multi-table support in OpenFlow 1.3 and newer hardware platforms like P4 [31] can make devirtualization more efficient. If multiple tables in a switch can be configured to in a pipeline to mirror the virtual network topology, then updating virtual switch tables can be directly mapped to updating physical tables. This can dramatically reduce compilation and rule update overhead. A complete exploration of this direction is part of our future work.

8 Proposed OpenFlow Extensions

The current language for communicating between CoVisor and its controllers is OpenFlow. We made this choice because OpenFlow is the current *lingua franca* of software-defined networks. Nevertheless, it is well-known that OpenFlow is not the ultimate SDN control protocol; both researchers and practitioners have been exploring extensions and revisions to the protocol for years. However, our use of OpenFlow in CoVisor has highlighted additional limitations that researchers might consider when revising the OpenFlow standard or when designing future protocols.

In particular, a single OpenFlow rule can only express *positive* properties of packets in a compact manner. For example, a single rule can forward a packet with type SSH out port 3, but it cannot forward a packet that does *not* have type SSH out port 3. This lack of expressiveness can be problematic if one would like to construct a hypervisor that allows controller A to choose to handle some traffic, while other traffic falls through to controller B. Such a situation is expressed naturally as $A \triangleright B$ in our system. However, if A chooses (during the course of operation) to control forwarding for packets that do *not* have type SSH, it can only do so by providing rules for all types of packets other than SSH packets. If OpenFlow supplied a *don't care* action (analogous to Pyretic's *pass-through* action [20]), controllers could generate just two rules to deal with such situations: a high-priority rule for SSH traffic with a don't care action and a lower priority rule that forwards all other traffic as desired. Of course, it would be possible for us to "hack" the OpenFlow protocol so controllers can transmit such information coded somehow, but hacking protocols in this fashion is brittle and leads to long-term software engineering nightmares.

9 Related Work

Slicing: Existing network hypervisors mostly focus on slicing; they target multi-tenancy scenarios in which each tenant operates on a disjoint subset, or *slice*, of the traffic [3, 4, 32, 33]. In contrast, CoVisor allows multiple controllers to collaborate on processing the same traffic.

Topology abstraction: Many projects studied the many-to-one case [20, 18, 19, 34]. Pyretic [20] explored the one-to-many case, but its implementation reactively installs micro-flow rules. CoVisor provides the first proactive compilation algorithm by leveraging symbolic analysis to build symbolic paths [21] and applying incremental sequential composition to generate the rules.

Composition: The parallel and sequential operators are proposed in the Frenetic project [5, 20], and the override operator is described in [15, 16, 35]. An incremental compilation algorithm for Frenetic policies is introduced in [16]. CoVisor is novel in using these operators to compose policies written on a variety of controller platforms, rather than just Frenetic. Furthermore, CoVisor takes advantage of the OpenFlow rules' explicit priorities; it uses a convenient algebra to calculate priorities for composed rules, thereby eliminating the need to build dependency graphs for rules and maintain scattered priority distributions [16]. Moreover, [16] only optimizes priority assignment for Frenetic policies; it is not a hypervisor to compose controllers, and does not have algorithms to compile topology virtualizations and optimizations by exploiting policy structures. Finally, it is an open problem to design a good interface for Frenetic to aid incremental update.

CoVisor is built upon our previous workshop paper [17], which presents the rule priority algebra for the parallel and sequential operators. This work includes new results on the algebra for the override operator, a proactive and incremental compilation algorithm for one-to-many virtualization, further optimization of compilation algorithms by exploiting policy structures, and a prototype implementation and evaluation.

Switch table type patterns: Table Type Patterns [36] and P4 [37] provide a syntax for describing flow table capabilities (e.g., fields that can be matched and modified). CoVisor uses this kind of information to build a customized data structure to optimize compilation. CoVisor's optimization technique differs from existing ways to index and accelerate multi-dimensional classifiers that don't know policy structures *a priori* [22, 23, 24, 38].

10 Conclusion

We present CoVisor, a compositional hypervisor that allows administrators to combine multiple controllers to collaboratively process a network's traffic. CoVisor uses a combination of novel algorithms and data structures to efficiently compile policies in an incremental manner. Evaluations on our prototype show that it is several orders of magnitude faster than a naive implementation, and we believe this is just the start of research on compositional hypervisors. There are many interesting future directions. In particular, extending existing and exploring new compilation techniques for multi-table support in compositional hypervisor setting is a very promising direction [31, 39]. First, this allows us to make efficient use of hardware capabilities and reduce the size of final policies for composition and devirtualization. Second, it introduces an incremental deployment path for hardware with OpenFlow 1.3 support as legacy applications written in OpenFlow 1.0 can run on top of CoVisor with CoVisor compiling them to OpenFlow 1.3.

Acknowledgements We thank Ronaldo Ferreira, Nanxi Kang, Srinivas Narayana, Mina Tahmasbi Arashloo, our shepherd Jeff Chase, and the anonymous NSDI reviewers for their feedback on earlier versions of this paper. This work is supported by NSF grant TC-1111520 and DARPA grant MRC-007692-001.

References

[1] "Ryu OpenFlow Controller." `http://osrg.github.io/ryu/`.

[2] "Floodlight OpenFlow Controller." `http://floodlight.openflowhub.org/`.

[3] A. Al-Shabibi, M. De Leenheer, M. Gerola, A. Koshibe, G. Parulkar, E. Salvadori, and B. Snow,

"OpenVirteX: Make your virtual SDNs programmable," in *ACM SIGCOMM HotSDN Workshop*, August 2014.

[4] R. Sherwood, G. Gibb, K. Yap, G. Appenzeller, M. Casado, N. McKeown, and G. Parulkar, "Can the production network be the testbed?," in *USENIX OSDI*, October 2010.

[5] N. Foster, M. J. Freedman, A. Guha, R. Harrison, N. P. Katta, C. Monsanto, J. Reich, M. Reitblatt, J. Rexford, C. Schlesinger, A. Story, and D. Walker, "Languages for software-defined networks," *IEEE Communications*, vol. 51, pp. 128–134, February 2013.

[6] S. Jain, A. Kumar, S. Mandal, J. Ong, L. Poutievski, A. Singh, S. Venkata, J. Wanderer, J. Zhou, M. Zhu, *et al.*, "B4: Experience with a globally-deployed software defined WAN," in *ACM SIGCOMM*, August 2013.

[7] C.-Y. Hong, S. Kandula, R. Mahajan, M. Zhang, V. Gill, M. Nanduri, and R. Wattenhofer, "Achieving high utilization with software-driven WAN," in *ACM SIGCOMM*, August 2013.

[8] Z. Qazi, C. Tu, L. Chiang, R. Miao, V. Sekar, and M. Yu, "SIMPLE-fying middlebox policy enforcement using SDN," in *ACM SIGCOMM*, August 2013.

[9] M. Moshref, M. Yu, R. Govindan, and A. Vahdat, "DREAM: dynamic resource allocation for software-defined measurement," in *ACM SIGCOMM*, August 2014.

[10] R. Wang, D. Butnariu, and J. Rexford, "OpenFlow-based server load balancing gone wild," in *Hot-ICE Workshop*, March 2011.

[11] B. Heller, S. Seetharaman, P. Mahadevan, Y. Yiakoumis, P. Sharma, S. Banerjee, and N. McKeown, "ElasticTree: Saving energy in data center networks.," in *USENIX NSDI*, April 2010.

[12] M. Al-Fares, S. Radhakrishnan, B. Raghavan, N. Huang, and A. Vahdat, "Hedera: Dynamic flow scheduling for data center networks.," in *USENIX NSDI*, April 2010.

[13] T. Benson, A. Anand, A. Akella, and M. Zhang, "MicroTE: Fine grained traffic engineering for data centers," in *ACM CoNEXT*, December 2011.

[14] A. R. Curtis, J. C. Mogul, J. Tourrilhes, P. Yalagandula, P. Sharma, and S. Banerjee, "DevoFlow: Scaling flow management for high-performance networks," in *ACM SIGCOMM*, August 2011.

[15] A. Gupta, L. Vanbever, M. Shahbaz, S. P. Donovan, B. Schlinker, N. Feamster, J. Rexford, S. Shenker, R. Clark, and E. Katz-Bassett, "SDX: A software defined internet exchange," in *ACM SIGCOMM*, August 2014.

[16] X. Wen, L. Li, C. Diao, X. Zhao, and Y. Chen, "Compiling minimum incremental update for modular SDN languages," in *ACM SIGCOMM HotSDN Workshop*, August 2014.

[17] X. Jin, J. Rexford, and D. Walker, "Incremental update for a compositional SDN hypervisor," in *ACM SIGCOMM HotSDN Workshop*, August 2014.

[18] N. Kang, Z. Liu, J. Rexford, and D. Walker, "Optimizing the one big switch abstraction in software-defined networks," in *ACM CoNEXT*, December 2013.

[19] Y. Kanizo, D. Hay, and I. Keslassy, "Palette: Distributing tables in software-defined networks," in *IEEE INFOCOM*, April 2013.

[20] C. Monsanto, J. Reich, N. Foster, J. Rexford, and D. Walker, "Composing software-defined networks," in *USENIX NSDI*, April 2013.

[21] P. Kazemian, M. Chang, H. Zeng, G. Varghese, N. McKeown, and S. Whyte, "Real time network policy checking using header space analysis," in *USENIX NSDI*, April 2013.

[22] P. Gupta and N. McKeown, "Packet classification using hierarchical intelligent cuttings," in *Hot Interconnects*, August 1999.

[23] S. Singh, F. Baboescu, G. Varghese, and J. Wang, "Packet classification using multidimensional cutting," in *ACM SIGCOMM*, August 2003.

[24] B. Vamanan, G. Voskuilen, and T. N. Vijaykumar, "EffiCuts: Optimizing packet classification for memory and throughput," in *ACM SIGCOMM*, August 2010.

[25] "Java Standard Library." `http://docs.oracle.com/javase/7/docs/api/`.

[26] "Concurrent-trees Library." `https://code.google.com/p/concurrent-trees/`.

[27] N. Handigol, B. Heller, V. Jeyakumar, B. Lantz, and N. McKeown, "Reproducible network experiments using container-based emulation," in *ACM CoNEXT*, December 2012.

[28] D. E. Taylor and J. S. Turner, “ClassBench: A packet classification benchmark,” in *IEEE INFOCOM*, March 2005.

[29] X. Jin, H. H. Liu, R. Gandhi, S. Kandula, R. Mahajan, M. Zhang, J. Rexford, and R. Wattenhofer, “Dynamic scheduling of network updates,” in *ACM SIGCOMM*, August 2014.

[30] C. Rotsos, N. Sarrar, S. Uhlig, R. Sherwood, and A. W. Moore, “OFLOPS: An open framework for OpenFlow switch evaluation,” in *Passive and Active Measurement Conference*, March 2012.

[31] L. Jose, L. Yan, G. Varghese, and N. McKeown, “Compiling packet programs to reconfigurable switches,” in *USENIX NSDI*, May 2015.

[32] T. Koponen, K. Amidon, P. Balland, M. Casado, A. Chanda, B. Fulton, I. Ganichev, J. Gross, N. Gude, P. Ingram, *et al.*, “Network virtualization in multi-tenant datacenters,” in *USENIX NSDI*, April 2014.

[33] R. Doriguzzi Corin, M. Gerola, R. Riggio, F. De Pellegrini, and E. Salvadori, “VeRTIGO: Network virtualization and beyond,” in *European Workshop on Software Defined Networks (EWSDN)*, October 2012.

[34] M. Casado, T. Koponen, R. Ramanathan, and S. Shenker, “Virtualizing the network forwarding plane,” in *Workshop on Programmable Routers for Extensible Services of Tomorrow*, November 2010.

[35] M. Canini, D. De Cicco, P. Kuznetsov, D. Levin, S. Schmid, S. Vissicchio, *et al.*, “STN: A robust and distributed SDN control plane,” in *Open Networking Summit*, March 2014.

[36] “OpenFlow Table Type Patterns 1.0.” `http://tinyurl.com/oebjbsf`.

[37] P. Bosshart, D. Daly, G. Gibb, M. Izzard, N. McKeown, J. Rexford, C. Schlesinger, D. Talayco, A. Vahdat, G. Varghese, and D. Walker, “P4: Programming protocol-independent packet processors,” *SIGCOMM CCR*, vol. 44, pp. 87–95, July 2014.

[38] N. Shelly, E. J. Jackson, T. Koponen, N. McKeown, and J. Rajahalme, “Flow caching for high entropy packet fields,” in *ACM SIGCOMM HotSDN Workshop*, August 2014.

[39] C. Schlesinger, M. Greenberg, and D. Walker, “Concurrent NetCore: From policies to pipelines,” in *ACM ICFP*, September 2014.

Compiling Packet Programs to Reconfigurable Switches

Lavanya Jose⋆, Lisa Yan⋆, George Varghese‡, Nick McKeown⋆
⋆*Stanford University,* ‡*Microsoft Research*

Abstract

Programmable switching chips are becoming more commonplace, along with new packet processing languages to configure the forwarding behavior. Our paper explores the design of a compiler for such switching chips, in particular how to map logical lookup tables to physical tables, while meeting data and control dependencies in the program. We study the interplay between Integer Linear Programming (ILP) and greedy algorithms to generate solutions optimized for latency, pipeline occupancy, or power consumption. ILP is slower but more likely to fit hard cases; further, ILP can be used to suggest the best greedy approach. We compile benchmarks from real production networks to two different programmable switch architectures: RMT and Intel's FlexPipe. Greedy solutions can fail to fit and can require up to 38% more stages, 42% more cycles, or 45% more power for some benchmarks. Our analysis also identifies critical resources in chips. For a complicated use case, doubling the TCAM per stage reduces the minimum number of stages needed by 12.5%.

1 Introduction

The Internet pioneers called for "dumb, minimal and streamlined" packet forwarding [11]. However, over time, switches have grown complex with the addition of access control, tunneling, overlay formats, etc., specified in over 7,000 RFCs. Programmable switch hardware called NPUs [10, 17] were an initial attempt to address changes. Yet NPUs, while flexible, are too slow: the fastest fixed-function switch chips today operate at over 2.5Tb/s, an order of magnitude faster than the fastest NPU, and two orders faster than a CPU.

As a consequence, almost all switching today is done by chips like Broadcom's Trident [9]; arriving packets are processed by a fast sequence of pipeline stages, each dedicated to a fixed function. While these chips have adjustable parameters, they fundamentally cannot be reprogrammed to recognize or modify new header fields. Fixed-function processing chips have two major disadvantages: first, it can take 2–3 years before new protocols are supported in hardware. For example, the VxLAN field [21]—a simple encapsulation header for network virtualization—was not available as a chip feature until three years after its introduction. Second, if the switch pipeline stages are dedicated to specific functions but only a few are needed in a given network, many of the switch table and processing resources are wasted.

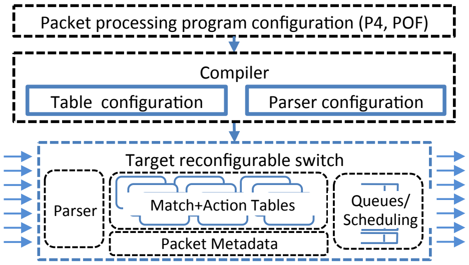


Figure 1: A top-down switch design.

A subtler consequence of fixed-function hardware is that networking equipment today is designed *bottom-up* rather than *top-down*. The designer of a new router must find a chip datasheet conforming to her requirements before squeezing her design bottom-up into a predetermined use of internal resources. By contrast, a top-down design approach would enable a network engineer to describe how packets are processed and adjust the sizes of various forwarding tables, oblivious to the the underlying hardware capabilities (Figure 1). Further, if the engineer changes the switch mid-deployment, she can simply install the existing program onto the new switch. The bottom-up design style is also at odds with other areas of high technology: for example, in graphics, the fastest DSPs and GPU chips [23, 28] provide primitive operations for a variety of applications. Fortunately, three trends suggest the imminent arrival of top-down networking design:

1. Software-Defined Networking (SDNs): SDNs [18, 24] are transforming network equipment from a vertically integrated model towards a programmable software platform where network owners and operators decide network behavior once deployed.

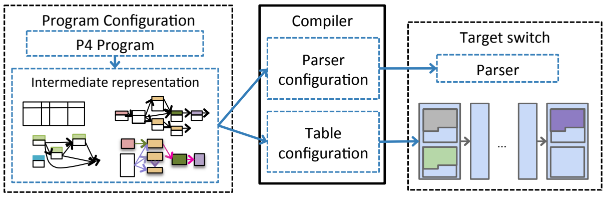


Figure 2: Compiler input and output.

2. Reconfigurable Chips: Emerging switch chip architectures are enabling programmers to reconfigure the packet processing pipeline at runtime. For example, the Intel FlexPipe [25], the RMT [8], and the Cavium XPA [4] follow a flexible *match+action* processing model that maintains performance comparable to fixed-function chips. Yet to accommodate flexibility, the switches have complex constraints on their programmability.

3. Packet Processing Languages: Recently, new languages have been proposed to express packet processing, like Huawei's Protocol Oblivious Forwarding [3, 27] and P4 [1, 2, 6]. Both POF and P4 describe how packets are to be processed abstractly—in terms of match+action processing—without referencing hardware details. P4 can be thought of as specifying control flow between a series of logical match+action tables.

With the advent of programmable switches and high-level switch languages, we are close to programming networking behavior top-down. However, a top-down approach is impossible without a *compiler* to map the high-level program down to the target switch hardware. This paper is about the design of such a compiler, which maps a given program configuration—using an intermediate representation called a Table Dependency Graph, or TDG (Section 2.1)—to a target switch. The compiler should create two items: a *parser configuration*, which specifies the order of headers in a packet, and a *table configuration*, which is a mapping that assigns the match+action tables to memory in a specific target switch pipeline (Figure 2). Previous research has shown how to generate parsing configurations [16]; the second aspect, table configuration, is the focus of this paper.

To understand the compilation problem, we first need to understand what a high-level packet processing language specifies and how an actual switch constrains feasible table configurations.

1.1 Packet Processing Languages

High-level packet processing languages such as P4 [6] must describe four things:

Abstract Switch Model: P4 uses the abstract switch model in Figure 1 with a programmable parser followed by a set of match+action tables (in parallel and/or in series) and a packet buffer.

Headers: P4 declares names for each header field, so the switch can turn incoming bit fields into typed data the programmer can reference. Headers are expressed using a parse graph, which can be compiled into a state machine using the methods of [16, 19].

Tables: P4 describes the *logical tables* of a program, which are match+action tables with a maximum size; examples are a 48-bit Ethernet address exact match table with at most 32,000 entries, or an 8K-entry table of 256-bit wide ACL matches.

Control Flow: P4 specifies the control flow that dictates how each packet header is to be processed, read, and modified. A compiler must map the program while preserving control flow; we give a more detailed example of this requirement in Section 2.1.

1.2 Characteristics of Switches

Once we have a high-level specification in a language, the compiler must work within the constraints of a target switch, which include the following:

Table sizes: Hardware switches contain memories that can be accessed in parallel and whose number and granularity are constrained.

Header field sizes: The width of the bus carrying the headers limits the size and number of headers that the switch can process.

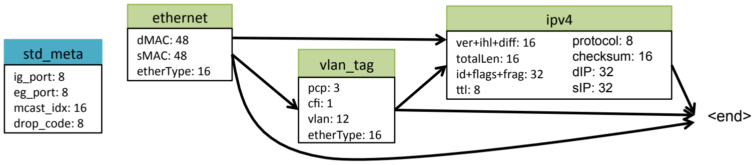
Matching headers: There are constraints on the width, format, and number of lookup keys to match against in each match+action stage.

Stage Diversity: A stage might have limited functionality; for example, one stage may be designed for matching IP prefixes, and another for ACL matching.

Concurrency: The biggest constraints often come from concurrency options. The three recent flexible switch ASICs (FlexPipe, RMT, XPA) are built from a sequential pipeline of match+action stages, with concurrency possible within each stage. A compiler must analyze the high-level program to find dependencies that limit concurrency; for example, a data dependency occurs when a piece of data in a header cannot be processed until a previous stage has finished modifying it. We follow the lead of Bosshart, et al. [8] and express dependencies using a TDG (Section 2.1).

1.3 Approach and Contributions

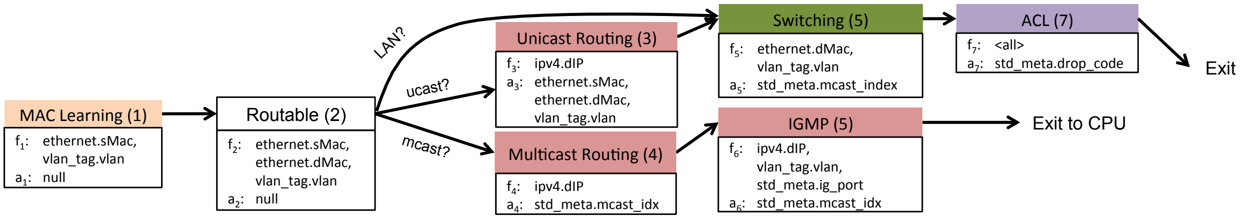
This paper is the first to define and systematically explore how to build a switch compiler by using abstractions to hide hardware details while capturing the essence required for mapping (Sections 2 and 3). Ideally we



(a) The parse graph specifies the order of the packet headers (green); the metadata (blue) is separate. All field names and lengths (in bits) are also specified.

Table name	l	Match type	e_l	n_l
MAC learning	1	exact	4000	(2)
Routable	2	exact	64	(3, 4, 5)
Unicast	3	prefix	2000	(5)
Multicast	4	prefix	500	(6)
Switching	5	exact	4000	(7)
IGMP	6	ternary	500	(to CPU)
ACL	7	ternary	1000	(exit)

(b) Each logical table l has a table name, maximum entry count e_l, and next table addresses n_l.



(c) The control flow program. Each table l has match fields f_l and modified fields a_l.

Figure 3: A packet processing program named L2L3 describing a simple L2/L3 IPv4 switch.

would like a switch-dependent front-end preprocessor, and a switch-independent back-end; we show how to relegate some switch-specific features to a preprocessor. We identify key issues for any switch compiler: table sizes, program control flow, and switch memory restrictions. In a sense, we are adapting instruction reordering [20], a standard compilation mechanism, to efficiently configure a packet-processing pipeline. We reinterpret traditional control and data dependencies [20] in a match+action context using a Table Dependency Graph (TDG).

A second contribution is to compare greedy heuristic designs to Integer Linear Programming (ILP) ones; ILP is a more general approach that lets us optimize across a variety of objective functions (e.g., minimizing latency or power). We analyze four greedy heuristics and several ILP solutions on two switch designs, FlexPipe and RMT. For the smaller FlexPipe architecture, we show that ILP can often find a solution when greedy fails. For RMT, the best greedy solutions can require 38% more stages, 42% more cycles, or 45% more power than ILP. We argue that with more constrained architectures and more complex programs (Section 6), ILP approaches will be needed.

A third contribution is exploring the interplay between ILP and greedy, given ILP's optimal mappings despite its longer runtime. For each switch architecture, we design a tailored greedy algorithm to use when a quick fit suffices. Further, by analyzing the ILP for the "tightest" constraints, we find we can improve the greedy heuristics. Finally, a sensitivity analysis shows that the most important chip constraints that limit mapping for our benchmarks are the *degree of parallelism* and *per-stage memory*.

We proceed as follows. Section 2 defines the mapping problem and TDG, Section 3 abstracts FlexPipe and RMT architectures, Section 4 presents our ILP formulation, and Section 5 describes greedy heuristics. Section 6 presents experimental results, and Section 7 describes sensitivity analysis to determine critical constraints.

2 Problem Statement

Our objective is to solve the table configuration problem in Figure 2. We focus on mapping P4 programs to FlexPipe and RMT, while respecting hardware constraints and program control flow. Since the abstract switch model in Figure 1 does not model realistic constraints such as concurrency limits, finite table space, and finite processing stages, the compiler needs two more pieces of information. First, the compiler creates a table dependency graph (TDG) from the P4 program to deduce opportunities for concurrency, described below. Second, the compiler must be given the physical constraints of the target switch; we consider constraints for specific chips in Section 3.

2.1 Table Dependency Graph

We describe program control flow using an example P4 program called L2L3. Figure 3 describes the program by showing three of the four items described in Section 1.1: headers, tables, and control flow. The fourth item, the abstract switch model, is described in Section 3.

Our L2L3 program supports unicast and multicast routing, Layer 2 forwarding and learning, IGMP snooping, and a small Access Control List (ACL) check. Figure 3a is a parse graph declaring three different header fields (Ethernet, IPv4, and VLAN) and metadata used

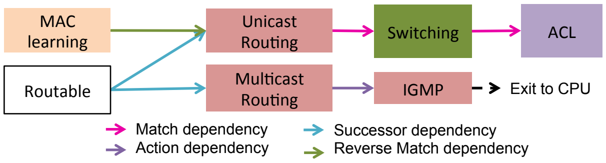


Figure 4: Table dependency graph for the L2L3 program.

Switch compiler	Traditional compiler
Match dependency	Read-After-Write
Action dependency	Write-After-Write
Successor dependency	Control dependence
Reverse-match dependency	Write-After-Read

Table 1: Mapping switch compiler dependencies to traditional compiler dependencies.

during processing. The features and control flow of the six logical tables in L2L3 are shown in Figure 3b and 3c.

Table l has attributes (f_l, e_l, a_l, n_l) that determine how a program should be allocated onto a target switch. A set of *match fields* f_l, from the packet header or metadata, are matched against e_l table entries. For example, the IPv4 Unicast routing table in L2L3 matches a 32-bit IPv4 destination address and holds up to 2,000 entries. In practice, table l may have much less than e_l entries, but the programmer provides e_l as an upper bound. Tables can have different *match types*: exact, prefix (longest prefix match), or ternary (wildcard). If the match type is ternary or prefix, the set f_l also specifies a bit mask. Based on the match result, the table performs actions on *modified fields* a_l and jumps to one of the tables specified in the set of *next table addresses*, n_l.

Figure 3c illustrates how header fields are processed by logical tables in an imperative control flow program. For example, the Unicast Routing table sets a new destination MAC address and VLAN tag before visiting the Switching table, which sets the egress port, and so on. The compiler must ensure that the matched and modified headers in each table correctly implement the control flow program.

We define a Table Dependency Graph (TDG) as a directed, acyclic graph (DAG) of the dependencies (edges) between the N logical tables (vertices) in the control flow. Dependencies arise between logical tables that lie on a common path through the control flow, where table outcomes can affect the same packet.

Figure 4 shows the TDG for our L2L3 program, which is generated directly from the P4 control flow and table description in Figure 3. From the next table addresses it is evident that some tables precede others in an execution pipeline; more precisely, Table A would precede Table B in an execution pipeline if there is a chain of tables $l_1, l_2, \ldots, l_k$ from A to B, where $l_1 \in n_A$, $l_2 \in n_{l_1}$, etc., and $B \in n_{l_k}$. If the result of Table A affects the outcome of Table B, we say that Table B has a *dependency* on Table A. In this case, there is an edge from Table A to B in the table dependency graph.

Different types of dependencies affect both the arrangement of tables in a pipeline and the pipeline latency.We present the three dependencies described in [8] and introduce a fourth below.

1. Match dependency: Table A modifies a field that a subsequent Table B matches.

2. Action dependency: Table A and B both change the same field, but the end-result should be that of the later Table B.

3. Successor dependency: Table A's match result determines whether Table B should be executed or not. More formally, there is a chain of tables $l_1, \ldots, l_k$ from A to B, where $l_1 \in n_A$, $l_2 \in n_{l_1}$, etc., and $B \in n_{l_k}$, such that every table $l_i \neq A$ in this chain is followed by B in each possible execution path. Additionally, there is a chain of next table adresses from A that does not go through B. For example, the Routable table's outcome determines whether Multicast Routing and IGMP will be executed. Thus, both have successor dependencies on Routable. On the other hand, IGMP does not have a successor dependency on Multicast Routing or vice-versa.

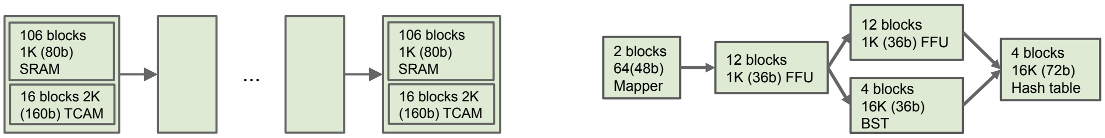
4. Reverse match dependency: Table A matches on a field that Table B modifies, and Table A must finish matching before Table B changes the field. This often occurs as in our example, where source MAC learning is an item that occurs early on, but the later Unicast table modifies the source MAC for packet exit.

Note that these dependencies roughly map to control and data dependencies in traditional compiler literature [5], where a match on a packet header field (or metadata) corresponds to a read and an action that updates a packet header (or metadata) corresponds to a write (Table 1).

While the TDG is strictly a multigraph, as there can be multiple dependencies between nodes, the mapping problem only depends on the strictest dependency that affects pipeline layout; the other dependencies can be removed to leave a graph. In summary, a TDG is a DAG of N logical tables (vertices) and dependencies (edges), where table $l \in \{1, \ldots, N\}$ has match fields, maximum match entries, modified fields, and next table addresses, denoted by f_l, e_l, a_l, and n_l, respectively.

3 Target Switches

The two backends we use—RMT [8] and Intel's FlexPipe [25]—represent real high-performance, programmable switch ASICs. Both conform to our abstract forwarding model (Figure 1) by implementing a pipeline of match+action stages and can run the L2L3 program in Figure 3. While both switches have different constraints, we can define hardware abstractions common to both chips: a pipeline DAG, memory types, and as-



(a) RMT switch as described in [8].

(b) Intel FlexPipe switch as described in [25].

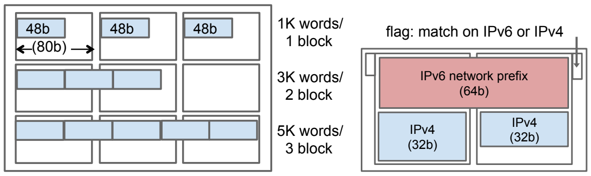
RMT						
s	Mem. Name	Type	m	b_m	w_m	d_m
1-32	SRAM	exact	1	106	80b	1K
	TCAM	ternary	2	16	40b	2K

(c) Memory information for RMT.

FlexPipe						
s	Mem. Name	Type	m	$b_{s,m}$	w_m	d_m
1	Mapper	exact	1	1	48b	64
2-3	FFU	ternary	2	12	36b	1K
4	BST	prefix	3	4	36b	16K
5	Hash Table	exact	4	4	72b	16K

(d) Memory information for FlexPipe.

Figure 5: Switch configurations for RMT and FlexPipe. The tuple (s,m) refers to memory type m ($m \in \{1,...,K\}$) in the stage indexed by $s \in \{1,...,M\}$. Each (s,m) has attributes $(b_{s,m}, w_m, d_m)$, where $b_{s,m}$ is the number of blocks of the m-th memory type, and each of these blocks can match d_m words (the "depth" of each block) of maximum width w_m bits.



(a) Word-packing for SRAM blocks with $(w_m, d_m) = (80\text{b}, 1000)$.

(b) Table-sharing.

Figure 6: Block layout features in different switches.

signment overhead. We describe these abstractions and switch-specific features, and highlight how our compiler represents each chip's constraints.

Pipeline Concurrency: We model the physical pipeline of each switch using a DAG of *stages* as shown in Figures 5a and 5b; a path from the i-th stage to the j-th stage implies that stage i starts execution before stage j. In the FlexPipe model (Figure 5b), the second Frame Forwarding Unit (FFU) stage and the Binary Search Tree (BST) stage can execute in parallel because there is no path between them.

Memory types: Switch designers decide in advance the allocation of different *memory blocks* based on programs they anticipate supporting. We abstract each memory block as having a *memory type* that supports various logical *match types* (Section 2.1). For example, in RMT, the TCAM allows ternary match type tables, while SRAM supports exact match only; in FlexPipe, FFU, hash tables, and BST memory types support ternary, exact, and prefix match, respectively.

Memory information for RMT and FlexPipe are in Tables 5c and 5d. We annotate the DAG to show the number, type and size of the memory blocks in each stage.

Assignment overhead: A table may execute actions or record statistics based on match results; these actions and statistics are also stored in the stage they are referenced. The number of blocks for action and statistics memory, collectively referred to as *assignment overhead*, is linearly dependent on the amount *match memory* a table has in a stage. In RMT, both TCAM and SRAM match memory store their overhead memory in SRAM; we ignore action and statistics memory in FlexPipe.

Combining entries: RMT allows a field to efficiently match against multiple words in the same memory block at a time, a feature we call *word-packing*. Different *packing formats* allow match entries to be efficiently stored in memory; for example, a packing format of 3 creates a *packing unit* that strings together two memory blocks and allows a 48b MAC address field to match against three MAC entries simultaneously in a 144b word (Figure 6a). FlexPipe only supports stringing together the minimum number of blocks required to match against one word, but does allow *table-sharing* in which multiple logical tables share the same block of SRAM or BST memory, provided the two tables are not on the same execution path. Table-sharing is shown in Figure 6b: since routing tables make decisions on either IPv4 or IPv6 prefixes, both sets of prefixes can share memory.

Per-stage resources: RMT uses three crossbars per stage to connect subsets of the packet header vector to the match and action logic. Matches in SRAM and TCAM, and actions all require crossbars composed of eight 80b-wide subunits for a total of 640 bits. A stage can match on at most 8 tables and modify at most 8 fields. There appears to be no analogous constraints for FlexPipe.

Latency: Generally, processing will begin in each pipeline stage as soon as data is ready, allowing for overlapping execution. However, logical dependencies restrict the overlap (Figure 7). In RMT, a match dependency means no overlap is possible, and the delay between two stages will be the latency of match and action in a stage: 12 cycles. Action dependent stages can have

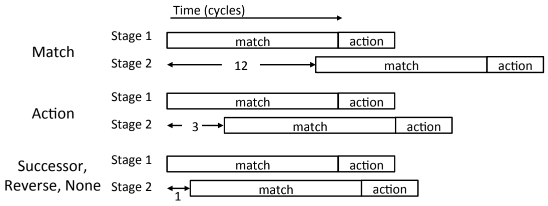


Figure 7: Dependency types and latency delays in RMT. In this figure, Table B in Stage 2 depends on Table A in Stage 1.

their match phases overlap, and so the minimum delay is 3 cycles between the stages. Successor and reverse-match dependencies can share stages, provided that tables can be run speculatively [8]. Note that even if there are no dependencies there is a one cycle delay between successive stages.

While RMT's architecture requires that match or action dependent tables be in strictly separate stages, FlexPipe's architecture resolves action dependencies at the end of each stage, and thus only match dependencies require separate stages. In summary, the compiler models specific switch designs abstractly using a DAG, multiple memory blocks per stage, constraints on packing, per-stage resources and latency characteristics. While we have described how to model RMT and FlexPipe (the only two currently published reconfigurable switches), new switches can be described using the same model if they use some form of physical match+action pipeline.

4 Integer Linear Programming

To build a compiler, we must map programs (parse graphs, table declarations and control flow) to target switches (modeled by a DAG of stages with per-stage resources) while maximizing concurrency and respecting all switch constraints. Because constraints are integer-valued (table sizes, crossbar widths, header vectors), it is natural to use Integer Linear Programming (ILP). If all constraints are linear constraints on integer variables and we specify an objective function (e.g., "use the least number of stages" or "minimize latency"), then fast ILP solvers (like CPLEX [12]) can find an optimal mapping.

We now explain how to encode switch and program constraints and specify objective functions. We divide the ILP-based compiler into a switch-specific preprocessor (for switch-specific resource calculation) and a switch-dependent compiler. We start with switch-independent common constraints.

4.1 Common Constraints

The following constraints are common to both switches:

Assignment Constraint: All logical tables must be assigned somewhere in the pipeline. For example, if a table l has $e_l = 5000$ entries, the total number of entries assigned to that logical table, or $W_{s,l,m}$ over all memory types m and stages s, should be at least 5000. Hence, we require:

$$\forall l : \sum_{s,m} W_{s,l,m} \geq e_l. \tag{1}$$

Capacity Constraint: For each memory type m, the aggregate memory assignment of table l to stage s, $U_{s,l,m}$, must not exceed the physical capacity of that stage, $b_{s,m}$:

$$\forall s,m : \sum_{l} U_{s,l,m} \leq b_{s,m}. \tag{2}$$

We define the assignment overhead as $\lambda_{m,l}$, which denotes the necessary number of action or statistics blocks required for assigning one match block of table l in memory type m. Thus the aggregate memory assignment is the sum of match memory blocks $\mu_{s,l,m}$ and assignment overhead blocks:

$$U_{s,l,m} = \mu_{s,l,m}(1 + \lambda_{l,m}).$$

Dependency Constraint: The solution must respect dependencies between logical tables. We use boolean variable $D_{A,B}$ to indicate whether table B depends on A, and the start and end stage numbers of any table l are denoted by S_l and E_l, respectively. If table B depends on A's results, then the first stage of B's entries, S_B, must occur after the match results of table A are known, which is at the earliest E_A (tables are allowed to span multiple pipeline stages):

$$\forall D_{A,B} > 0 : \quad E_A \leq S_B. \tag{3}$$

If A must completely finish executing before B begins (e.g., match dependencies), then the inequality in Equation 3 becomes strict.

4.2 Objective Functions

A key advantage of ILP is that it can find an optimal solution for an objective function. In the remainder of the paper we focus our attention on three objective functions.

Pipeline stages: To minimize the number of pipeline stages a program uses, σ, we ask ILP to minimize:

$$\min \ \sigma, \tag{4}$$

where for all stages s:

$$\text{If} \sum_{l,m} U_{s,l,m} > 0 : \qquad \sigma \geq s.$$

Latency: We can alternatively pick an objective function to minimize the total pipeline latency, which is more involved. Consider RMT, in which match and action dependencies both affect when a pipeline stage can start (whereas successor and reverse-match dependencies do not affect when a stage starts). If a table in stage s has a match or action dependency on a table in stage s', then s' cannot start until 12 or 3 clock cycles, respectively, after s. Building on how we expressed dependencies in Equation 3, we assign stage s a start time, t_s, where t_s is strictly increasing with s. Now consider two tables A and B, and assume B has a match dependency (i.e. 12 cycle wait) on table A. E_A is the last stage A resides in, and S_B is the first stage B resides in. We constrain S_B as follows:

$$t_{E_A} + 12 \le t_{S_B}.$$

We write the same constraints for all pairs of tables with action dependencies (3 cycle wait). Then we minimize the start time of the last stage, stage M:

$$\min \ t_M. \tag{5}$$

Power: Our third objective function minimizes power consumption by minimizing the number of active memory blocks, and where possible uses SRAM instead of TCAM. The objective function is therefore as follows:

$$\min \ \sum_m g_m\Big(\sum_{s,l} U_{s,l,m}\Big), \tag{6}$$

where $g_m(\cdot)$ returns the power consumed for memory type m.

4.3 Switch-Specific Constraints

Our ILP model requires switch-specific constraints, and we push as many details as possible to our preprocessor.

RMT: We start with RMT's ability to pack memory words together to create wider matches. Recall from Section 3 that a packing format p packs together p words in a single wide match; $B_{l,m,p}$ specifies the number of memory type m blocks required for packing format p of table l. While $B_{l,m,p}$ is precomputed by the preprocessor from the widths of the table entries and memory blocks, the ILP solver decides the number of packing units $P_{s,l,m,p}$ for each stage. We can thus find the number of match memory blocks $\mu_{s,l,m}$ and number assigned entries $W_{s,l,m}$ for each stage:

$$\mu_{s,l,m} = \sum_{p=1}^{p_{max}} P_{s,l,m,p} B_{l,m,p}.$$

$$W_{s,l,m} = \sum_{p=1}^{p_{max}} P_{s,l,m,p}(p \cdot d_m),$$

where $p \cdot d_m$ is the number of table l's entries that can fit in a single packing unit of format p in memory type m.

Per-Stage Resource Constraints: We must incorporate RMT-specific constraints such as the input action and match crossbars. The preprocessor can compute the number of input and action subunits needed for a logical table as a function of the width of the fields on which it matches or modifies, respectively.

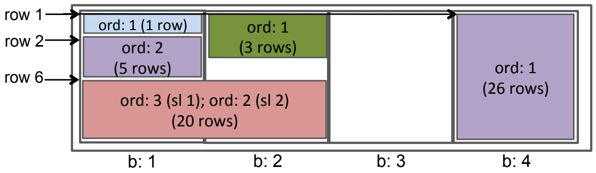


Figure 8: FlexPipe table sharing in detail. The pink table occupies the first two memory blocks, but different sets of tables share the first two memory blocks.

FlexPipe: FlexPipe can share memory blocks at a finer granularity than RMT, and so we need to preprocess the constraints differently for FlexPipe.

To support configurations as in Figure 8, we need to know which rows within a set of blocks are assigned to each logical table. This is because multiple tables can share a block, and different blocks associated with the same table can have very different arrangement of tables, such as blocks 1 and 2 assigned to the pink table.

Note that this issue does not arise in RMT; all memory blocks that contain a logical table will be uniform, and a solution can be rearranged to group together all memory blocks of a particular table assignment in a stage. We thus index the memory blocks $b \in 1, \ldots, b_{s,m}$, where $b_{s,m}$ is the maximum number of blocks of type m in stage s.

The solver decides how many logical table entries to assign to each block in each stage. For the remainder of this discussion, we differentiate between logical table entries and physical memory block entries by referring to the latter as rows, where row 1 and row e_m are the first and last rows, respectively, of a block of memory type m.

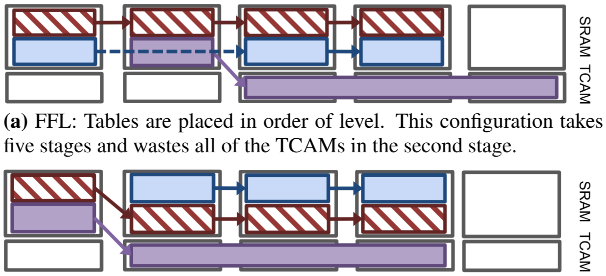
For table l assigned to start in the bth block of memory type m, we use the variable $\hat{r}_{l,m,b}$ to denote the starting row, and the variable $r_{l,m,b}$ to denote the number of consecutive rows that follow. [1]

To make sure rows do not overlap within a block, we constrain their placement by introducing the notion of *order*. Order is defined by the variable $\theta \in \{1, \ldots, \theta_{max}\}$, where θ_{max} is the maximum number of logical tables that can share a given memory block. In Figure 8, the light blue assignment has order $\theta = 1$, because it has the earliest row assignment. We define two additional variables, $\hat{\rho}_{m,b,\theta}$ and $\rho_{m,b,\theta}$, the start row and the number of rows of table with order θ, and we prevent overlaps by constraining the assignment as follows.

If θ-th order is assigned:

$$\hat{\rho}_{m,b,\theta-1} + \rho_{m,b,\theta} \le \hat{\rho}_{m,b,\theta}$$

[1] Note that if a second table, l', has entries in an adjacent block b', but the entries are wide and overflow into block b, $\hat{r}_{l',m,b} = 0$ because the starting row for l' was not assigned in this block; similarly, $r_{l,m,b}$ is irrelevant.



(**a**) FFL: Tables are placed in order of level. This configuration takes five stages and wastes all of the TCAMs in the second stage.

(**b**) FFLS: The first purple table with a large ternary table following it is placed first, even though the blue table has more match dependencies following it. This configuration uses only four stages.

Figure 9: Multiple-metric heuristics. A toy RMT example where a table with a single, dependent large ternary table must be placed before a table with a longer dependency chain.

To calculate the assignment constraint (Equation 1), the total number of words assigned to table l in stage s is:

$$W_{s,l,m} = \sum_{b=1}^{b_{s,m}} r_{l,m,b}.$$

where $r_{l,m,b}$ denotes the number of rows assigned for table l in all orders θ in block b of memory type m.

While the capacity constraint in Equation 2 is per stage, in FlexPipe we must also implement a capacity constraint per block. We restrict the number of rows we can assign to a block by checking the last row of the last order, θ_{max}:

$$\hat{\rho}_{m,b,\theta_{max}} + \rho_{m,b,\theta_{max}} \leq d_m.$$

Dependency Constraints: Fortunately, the dependency analysis is similar to RMT in Section 4.3, with the additional feature that only match dependencies require a strict inequality; action, successor, and reverse-match dependencies can be resolved in the same stage.

Objectives: Since FlexPipe has a short pipeline, we minimize the number of blocks used across all stages.

5 Greedy Heuristics

Since a full-blown optimal ILP algorithm takes a long time to run, we also explored four simpler greedy heuristics for our compiler: First Fit by Level (FFL), First Fit Decreasing (FFD), First Fit by Level and Size (FFLS), and Most Constrained First (MCF). All four greedy heuristics work as follows: First, sort the logical tables according to a metric. For each logical table in sorted order, pick the first set of memory blocks in the first pipeline stage the table can fit in without violating any capacity constraints, dependencies, or switch-specific resources. If it cannot fit, the heuristic finds the next available memory blocks, in the same stage or a subsequent stage. Like ILP, we leave switch-specific resource calculation like crossbar units and packing formats to a preprocessor. A heuristic terminates when all tables have been assigned or when it runs out of resources.

5.1 Ordering tables

The quality of the mapping depends heavily on the sort order. Three sorting metrics seem to matter most in our experiments, described in more detail below.

Dependency: Tables that come early in a *long dependency chain* should be placed first because we need at least as many stages left as there are match/action dependencies. We thus define the *level* of a table to be the number of match+action dependencies in the longest path of the TDG from the table to the end.

Word width: In RMT, tables with wide match or action words use up a large fraction of the *fixed resources* (action/input crossbars) and should be prioritized; they may not have room if smaller tables are assigned first. In FlexPipe, tables with larger match word width should be assigned first because there is less memory per stage.

Memory Types: While blocks with memory types like TCAM, BST, and FFU can also fit exact-match tables, exact memory like SRAM is generally more abundant than flexible memory due to switch costs. Thus in FlexPipe, heuristics should prioritize the assignment of more *restrictive tables*, or tables that can only go in ternary or prefix memories; otherwise, assigning exact match tables to flexible memories first can quickly lead to memory shortage. In RMT, restrictive tables go into TCAM, available in every stage. But large TCAM tables in a long dependency chain push back tables that follow them. So we should prioritize tables that imply high TCAM usage in their dependency chain.

Single-metric heuristics: Two of our greedy heuristics sort on a single metric: FFL is inspired by bin packing with dependencies [15] and sorts on table level, where tables at the head of long dependency chains are placed earlier. FFD is based on the First Fit Decreasing Heuristic for bin packing [15]. In our case, we prioritize tables that have wider action or match words and consequently use more action or input crossbar subunits. This heuristic should work well when fixed switch resources—and not program table sizes—are the limiting factor.

Multi-metric heuristics: Some programs fit well if we consider only one metric: if there are plenty of resources at each stage, we need only worry about long dependency chains. Our next two heuristics sort on multiple metrics. Sometimes being greedy on just one metric might not work, as shown in Figure 9: here, our first multi-metric heuristic FFLS incorporates dependencies and TCAM usage, where tables with larger TCAM tables in their dependency chains are assigned earlier.

Our other multi-metric heuristic, MCF, is motivated by FlexPipe's smaller pipeline with more varied memory types. We pick the "most constrained" table first: a table restricted to a particular memory type and with a

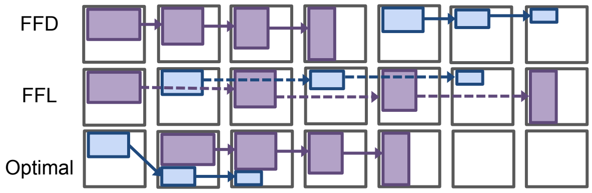
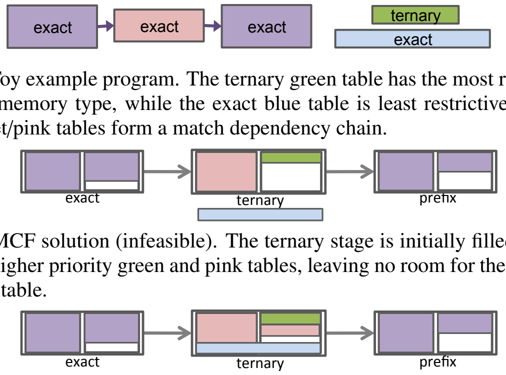


Figure 10: Greedy performing much worse than ILP (RMT). In this toy example, the blue and purple tables form separate match dependency chains. The initial blue table in the optimal mapping is narrower and has a lower level than the purple table, counterintuitive to both FFD and FFL metrics.



(a) Toy example program. The ternary green table has the most restrictive memory type, while the exact blue table is least restrictive. The violet/pink tables form a match dependency chain.

(b) MCF solution (infeasible). The ternary stage is initially filled with the higher priority green and pink tables, leaving no room for the wider blue table.

(c) Optimal solution. The pink table is split across ternary blocks, leaving enough room for the blue table.

Figure 11: Greedy performing much worse than ILP (FlexPipe).

high level should have higher priority. Ties are broken by placing the table with wider match words first. FFL and FFD which ignore the memory type do not work well for FlexPipe, which does not have uniform memory layout per stage like RMT; for example, ternary match tables can only go in stages 2 or 3 in FlexPipe.

Variations: Each of the basic four heuristics has two variants: by default, an exact match table spills into TCAMs if it runs out of SRAMs in a stage. Our first variant prevents the spillage to preserve the TCAMs for ternary tables. Second, by default, when we reserve space for a TCAM table, we do not reserve space in SRAM to hold the associated action data, which means we may run out of SRAM and not be able to use the TCAM. Our second variant sets aside SRAM for action memory from yet-to-be allocated ternary tables; in our experiments we fix the amount to be 16 SRAMs.

There can be cases where the best combination of metrics is unclear, as in Figure 10 for RMT and Figure 11 for FlexPipe. Our experiments in Section 6 seek the right combination of metrics for an efficient greedy compiler.

6 Experiments

We tested our algorithms by compiling the four benchmark programs listed in Table 2 for the RMT and FlexPipe switches. The benchmarks are in Table 2: L2L3-Simple, a simple L2/L3 program with large tables; L2L3-Mtag, which is L2L3-Simple plus support for the mTag toy example described in [7]; L2L3-Complex, a complex L2/L3 program for an enterprise DC aggregation router with large host routing tables, and L3DC, which is a program for Layer 3 switching in a smaller enterprise DC switch. Differently sized, smaller variations of the L2L3-Simple and L2L3-Mtag programs are used for FlexPipe. L2L3-Complex and L3DC cannot run on Flexpipe because the longest dependency chain for each program needs 9 and 6 stages respectively, exceeding FlexPipe's 5-stage pipeline.

Name	Switch	N	Dependencies		
			Match	Action	Other
L2L3 -Complex	RMT	24	23	2	10
L2L3 -Simple	RMT	16	4	0	15
	FlexPipe	13	12	0	4
L2L3 -Mtag	RMT	19	6	1	16
	FlexPipe	11	9	1	3
L3DC	RMT	13	7	3	1

Table 2: Logical program benchmarks for RMT and Flexpipe. N is the number of tables.

ILP: We used three ILP objective functions for RMT: number of stages (*ILP-Stages*), pipeline latency (*ILP-Latency*), and power consumption (*ILP-Power*). For FlexPipe, since we struggle to fit the program, we simply looked for a feasible solution that fit the switch. All of our ILP experiments were run using IBM's ILP solver, CPLEX. [2]

Greedy heuristics: For each RMT program, we ran all four greedy heuristics (FFD, FFL, FFLS, MCF). We also ran the variant that set aside 16 SRAM blocks for ternary action memory (labeled as FFD-16, etc.) and a combination of the two variants to also avoid spilling exact match tables into TCAM (labeled as FFD-exact16, etc.). For each FlexPipe program, we simply ran the greedy heuristic MCF. The other three heuristics do not combine enough metrics to fit either of our FlexPipe benchmarks.

All of our experiments were run on an Amazon AWS EC2 c3.4xlarge instance with 16 processor cores and 30 GB of memory. For FlexPipe, we generated 20 and 10 versions of the L2L3-Simple and L2L3-Mtag programs, respectively, with varying table sizes and checked how many greedy and ILP mappings fit the switch (Table 3). For RMT, we compiled every program 10 times for each of the greedy heuristics and the ILP objective functions

[2] CPLEX has a *gap tolerance* parameter, which sets the acceptable gap between the best integer objective and the current solution's objective. For ILP-Stage, we required zero-gap tolerance. For ILP-Latency and ILP-Power, we set the gap tolerance to be within 70% and 5%, respectively, of the best integer value; we found that lower gaps highly increased runtime with little improvement in objective value.

Solver	L2L3-Simple	L2L3-Mtag
	% solved	% solved
MCF	75	60
ILP	75	80

Table 3: Benchmark results for 5-stage FlexPipe.

Solver	L2L3-Complex			
	St.	Lat.	Pwr	RT.
FFD	22	135	4.98	0.25
FFD-16	21	135	5.51	0.27
FFD-exact16	21	135	4.62	0.27
FFL	19	131	5.61	0.25
FFL-16	19	131	6.09	0.27
FFL-exact16	17	132	4.61	0.24
FFLS	19	130	5.66	0.33
FFLS-16	19	130	6.42	0.35
FFLS-exact16	17	131	4.66	0.32
MCF	20	132	4.67	0.26
MCF-16	19	132	6.43	0.27
MCF-exact16	18	132	4.67	0.25
ILP-Latency	32	104	7.78	233.84
ILP-Stages	16	131	6.66	12.13
ILP-Power	32	131	4.44	147.10

Table 4: Benchmark results for RMT for L2L3-Complex. All greedy heuristics and variants are shown (St: number of stages occupied, Lat.: Latency [cycles], Pwr.: Power [Watts], RT.: Time to run solver [secs]).

and report the medians of the number of stages used, pipeline latency, and power consumed for each algorithm. We show in detail L2L3-Complex results in Table 4. To facilitate presentation, for all other programs we display results for ILP and the '-exact16' greedy variant only (Table 5), since this variant generally tended to have better stage and power usage than other greedy heuristics.

7 Analysis of Results

We analyze Tables 3, 4 and 5 for major findings. A salient observation is that the MCF heuristic for FlexPipe fits 16 out of the 20 versions of L2L3Simple. For some programs where the heuristic could not fit, it was difficult to manually analyze the incomplete solution for feasibility. However, ILP can both detect infeasible programs and find a fitting when feasible (assuming match tables are reasonably large,[3] e.g., occupy at least 5% of a hash table memory block.)

Another important observation for reconfigurable chips where one can optimize for different objectives, is that the best greedy heuristic can perform 25% worse on the objectives than ILP; for example, the optimal 104 cycle latency for ILP in the second column of Table 4 is far better than the best latency of 130 cycles by FFLS. A detailed comparison follows.

7.1 ILP vs Greedy

The following observations can be made after closely comparing ILP and greedy solutions in Figure 12.

1. Global versus local optimization: For the L2L3-Complex use case (Figure 12a), even the best greedy heuristic FFL-exact16 takes 17 stages, while ILP takes only 16 stages. Figures 12c and 12d show FFL-16 and ILP solutions, respectively. ILP breaks up tables over stages to pack them more efficiently, whereas greedy tries to assign as many words as possible in each stage per table, eventually wasting some SRAMs in some stages and using up more stages overall

In switch chips with shorter pipelines than RMT's, this could be the difference between fitting and not fitting. If all features in a program are necessary, then infeasibility is not an option. Unlike register allocation, there is no option to "spill to memory"; on the other hand, the longer runtime for ILP may be acceptable when adding a new router feature. Therefore it seems very likely that programmers will resort to optimal algorithms, such as ILP, when they really need to squeeze a program in.

2. Greedy poor for pipeline latency: Our greedy heuristics minimize the stages required to fit the program, and are good at minimizing power—the best greedy is only 4% worse than optimal (for L2L3-Complex, FFL-exact16 consumes 4.61W, versus ILP's 4.44W); technically, this is true only because the '-exact' variant avoids using power hungry TCAMs. But greedy heuristics are much worse for pipeline latency; minimizing latency with greedy algorithms will require improved heuristics.

7.2 Comparing greedy heuristics

1. Prioritize dependencies, not table sizes: In L2L3-Mtag, both FFL and FFLS assign the exact-match tables in the first stage, but differ in how they assign ternary tables. FFLS prioritizes the larger ACL table over the IPv6-Prefix and IPv6-Fwd tables, which are early tables in the long red dependency chain in Figure 13a. As a result, the IPv6-Prefix and IPv6-Fwd tables cannot start until stages 16 and 17, and FFLS ends up using two more stages than FFL. Though FFLS prioritizes large TCAM tables and avoids the problem discussed in Figure 9, it is not sophisticated enough to recognize other opportunities for sharing stages between dependency chains.

2. Sorting metrics matter: FFD results show that incorrect sorting order can be expensive (22 stages versus the optimal 16 for L2L3-Complex). We predict that FFD will only be useful for use cases with many wide logical tables or more limited per-stage switch resources, neither of which was a limiting factor in our experiments.

3. Set aside SRAM for TCAM actions: The '-16' vari-

[3] The minimum table size constraint helps us scale the ILP to handle FlexPipe, where a table can be assigned any number of rows in each memory block. Since the size of logical tables and memory blocks are at least on the order of hundreds, it seems reasonable to impose a minimum match table size of at least a hundred in these memory blocks.

Solver	L2L3-simple				L2L3-Mtag				L3DC			
	St.	Lat.	Pwr	RT.	St.	Lat.	Pwr	RT.	St.	Lat.	Pwr	RT.
FFD-exact16	21	64	7.54	0.18	22	75	7.65	0.21	7	88	2.34	0.08
FFL-exact16	19	55	7.55	0.19	19	66	7.66	0.21	7	88	2.34	0.08
FFLS-exact16	20	64	7.88	0.23	21	75	8.10	0.27	7	88	2.34	0.12
MCF-exact16	19	55	7.54	0.18	19	66	7.65	0.21	7	88	2.34	0.09
ILP-Latency	32	51	9.18	2.22	32	53	9.65	3.62	32	62	3.21	23.16
ILP-Stages	19	55	7.52	2.57	19	72	9.62	3.52	7	88	2.46	1.88
ILP-Power	32	62	7.55	2.27	32	71	7.63	2.53	9	86	2.34	1.87

Table 5: Benchmark results for 32-stage RMT for L2L3-simple, L2L3-Mtag, and L3DC. See Table 4 for units.

ation of our greedy heuristics estimates the number of SRAMs needed for ternary tables (for their action memory) in each stage and blocks them off when initially assigning SRAMs to exact match tables. Our experiments show that this local optimization usually avoids having to move a ternary table to a new stage because it doesn't have enough SRAMs for action memory.

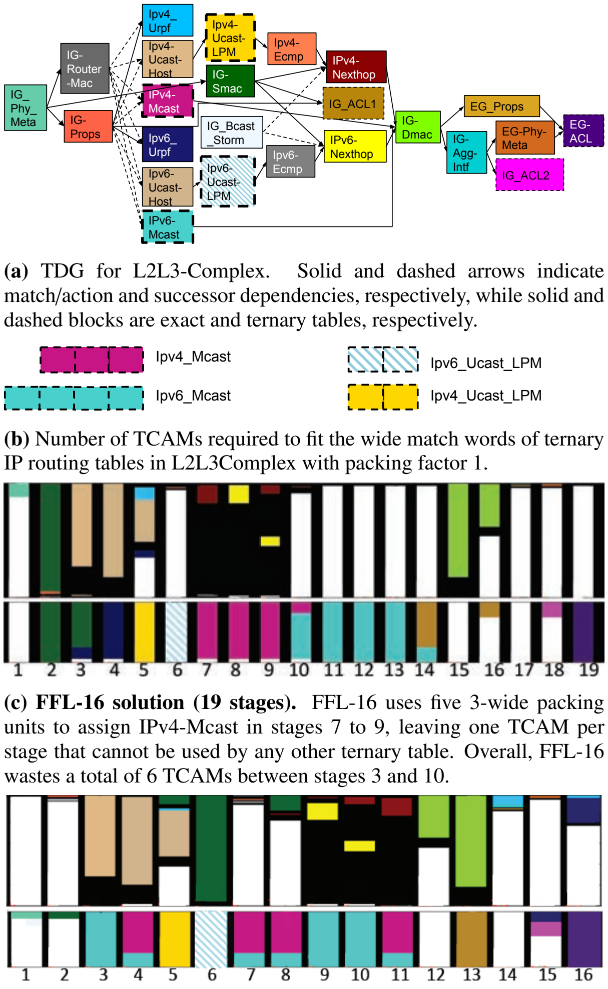
7.3 Sensitivity Experiments

In this section, we analyze ILP solutions by ignoring and relaxing various constraints in order to improve the running time of ILP and the optimality of greedy heuristics. We run these ILP experiments for our most complicated use case (L2L3_Complex) on the RMT chip, for two different objectives: minimum stages and minimum pipeline latency. For reference, the original ILP yields solution 16 stages in 12.13s and 104 cycles in 233.84s, respectively, for the two objectives.

To improve ILP runtime, we measure how long the ILP solver takes while ignoring or relaxing each constraint, which is a proxy for how hard it is to fit programs in the switch. This help us identify constraints that are currently "bottlenecks" in runtime for the ILP solver and also helps us understand how future switches can be designed to expedite ILP-based compilation.

We identify candidate metrics for greedy heuristics to optimize a given objective by ignoring constraints and identifying which have a significant impact on the quality of the solution. Our experiments also help identify the critical resources needed in the chip for typical programs, so chipmakers can design for better performance.

Sensitivity Results for ILP runtime: First, sizing particular resources can speed up IBM's ILP solver, CPLEX; for example, increasing the width of SRAM blocks by 37.5% (from 80b to 110b) reduces ILP runtime from 12.13s to 7.1s when minimizing stages. ILP run time is reduced considerably if action memory is not allocated, and this leads to a simple way to accelerate ILP: We first ran a greedy solution to estimate action memory needed per stage which is then set aside. We then ran the ILP without fitting the action memory, and finally added the action memory at the end. For minimizing pipeline latency, this reduced the ILP runtime from 233.84 sec-



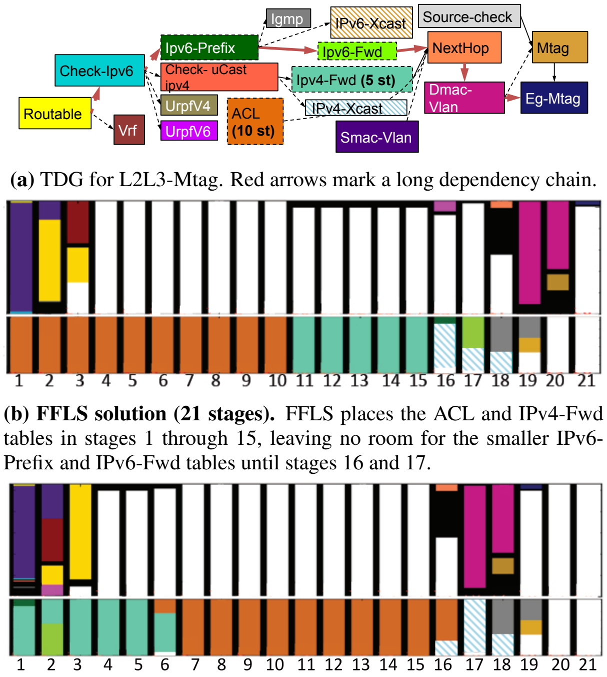
(a) TDG for L2L3-Complex. Solid and dashed arrows indicate match/action and successor dependencies, respectively, while solid and dashed blocks are exact and ternary tables, respectively.

(b) Number of TCAMs required to fit the wide match words of ternary IP routing tables in L2L3Complex with packing factor 1.

(c) FFL-16 solution (19 stages). FFL-16 uses five 3-wide packing units to assign IPv4-Mcast in stages 7 to 9, leaving one TCAM per stage that cannot be used by any other ternary table. Overall, FFL-16 wastes a total of 6 TCAMs between stages 3 and 10.

(d) ILP solution (16 stages). ILP utilizes all TCAMs in stages 4, 7, 8, and 11 by sharing the TCAMs between IPv4-Mcast (four 3-wide packing units) and IPv6-Mcast (one 4-wide packing unit).

Figure 12: FFL-16 and ILP solutions for L2L3-Complex. In assigning packing units to the ternary IP routing tables, FFL-16 locally maximizes the number of words per stage whereas ILP optimizes over a set of stages. Each stage has 106 SRAMs (top row) and 16 TCAMs (bottom row) and is colored according to the amount of match memory assigned to each logical table in the program TDG; all action memory is colored in black.



(a) TDG for L2L3-Mtag. Red arrows mark a long dependency chain.

(b) FFLS solution (21 stages). FFLS places the ACL and IPv4-Fwd tables in stages 1 through 15, leaving no room for the smaller IPv6-Prefix and IPv6-Fwd tables until stages 16 and 17.

(c) FFL solution (19 stages). FFL prioritizes the IPv6-Prefix and IPv6-Fwd tables and fits them in stages 1 and 2, allowing earlier assignment of Nexthop and other dependent tables. As a result, FFL needs two stages less than FFLS.

Figure 13: FFLS and FFL solutions for L2L3-Mtag. FFL prioritizes dependencies over table sizes and uses two fewer stages than FFLS.

onds to 66.29 seconds on average, without compromising our objective.

Sensitivity Results for optimality of greedy heuristics: We discovered that the dependency constraint for ILP has the largest impact on the minimum stages objective. If we remove dependencies from the TDGs, we can reduce the number of stages used from 16 to 13 and pipeline latency by 2 cycles. This explains why greedy heuristics focusing on the dependency metric (i.e., FFL and FFLS) do particularly well. Ignoring other constraints (like resource constraints) makes no difference to the number of stages used or latency. In addition, relaxing various resource constraints showed that some resources impact fitting more than others. For example, doubling the number of TCAM blocks per stage reduced the number of stages needed from 16 to 14. But doubling the number (or width) of crossbars made no difference. This explains why our FFD greedy heuristic (which focuses on non-limiting resources in the RMT switch) performs worse than other algorithms.

Lessons for chipmakers: Our results above indicate that chipmakers can improve turnaround for optimal ILP compilers by carefully selecting memory width. Moreover, if flexible memory is a rare resource, then increasing a non-limiting resource like crossbar complexity will not improve performance.

8 Related Work

Compiling packet programs to reconfigurable switches differs from compiling to FPGAs [26], other spatial architectures such as PLUG [14] or to NPUs [13]. We focus on *packing* match+action tables into memories in *pipelined stages* while satisfying dependencies. Nowatzki, et al. [22] develops an ILP scheduler for a spatial architecture that maps instructions entailed by program blocks to hardware, by allocating computational units for instructions and routing data between units. The corresponding problems for reconfigurable switches—assigning action units and routing data among the packet header, memories and action units—are less challenging once we have a table placement (and not in the scope of this paper.) NPUs such as the IXP network processor architecture [17] have multithreaded packet processing engines that can be pipelined. Approaches like that of Dai, et al. [13] map a sequential packet processing application into pipelined stages. However the processing engines have a large shared memory; thus NPU compilers do not need to address the problem of packing logical tables into physical memories.

9 Conclusion

We define the problem of mapping logical tables in packet processing programs. We evaluate greedy heuristics and ILP approaches for mapping logical tables on realistic benchmarks. While fitting tables is the main criterion, we also compute how well solvers minimize pipeline latency on the long RMT pipeline. We find that for RMT, there are realistic configurations where greedy approaches can fail to fit and need up to 38% more memory resources on the same benchmark. Three situations when ILP outperforms greedy are when there are *multiple conflicting metrics*, *multiple memory types* and *complicated objectives*. We believe future packet programs will get more complicated with more control flows, more different size tables, more dependencies and more complex objectives, arguing for an ILP-based approach. Further, sensitivity analysis of critical ILP constraints provides insight into designing fast tailored greedy approaches for particular targets and programs, marrying compilation speed to optimality.

Acknowledgements:

We thank Pat Bosshart and Dan Daly for many hours spent helping us understand RMT and Flexpipe. Many thanks also to Ravindra Sunkad and Changhoon Kim for providing us with detailed use cases. This material is based upon work supported by the National Science Foundation Graduate Research Fellowship Program under Grant No. DGE-114747.

References

[1] P4 language spec version 1.0.0-rc2. `http://www.p4.org/spec/p4-latest.pdf`. Accessed: 2014-09-22.

[2] P4 website. `http://www.p4.org/`. Accessed: 2014-09-22.

[3] Protocol oblivious forwarding (pof) website. `http://www.poforwarding.org/`. Accessed: 2014-09-22.

[4] Xpliant packet architecture (xpa) press release. `http://www.cavium.com/newsevents-Cavium\ \-and-XPliant-Introduce-a-Fully-Programmable\ \-Switch-Silicon-Family.html`. Accessed: 2014-09-22.

[5] Appel, A. W. *Modern compiler implementation in C*. Cambridge university press, 1997.

[6] Bosshart, P., Daly, D., Gibb, G., Izzard, M., McKeown, N., Rexford, J., Schlesinger, C., Talayco, D., Vahdat, A., Varghese, G., and Walker, D. P4: Programming protocol-independent packet processors. *SIGCOMM Comput. Commun. Rev. 44*, 3 (July 2014), 87–95.

[7] Bosshart, P., Daly, D., Izzard, M., McKeown, N., Rexford, J., Talayco, D., Vahdat, A., Varghese, G., and Walker, D. Programming protocol-independent packet processors. *CoRR abs/1312.1719* (2013).

[8] Bosshart, P., Gibb, G., Kim, H.-S., Varghese, G., McKeown, N., Izzard, M., Mujica, F., and Horowitz, M. Forwarding metamorphosis: Fast programmable match-action processing in hardware for SDN. In *Proceedings of the ACM SIGCOMM 2013 conference on SIGCOMM* (2013), ACM, pp. 99–110.

[9] Broadcom Corporation. *Broadcom BCM56850 StrataXGS® Trident II Switching Technology*. Broadcom, 2013.

[10] Cisco Systems. *Deploying Control Plane Policing*. Cisco White Paper, 2005.

[11] Clark, D. The design philosophy of the darpa internet protocols. In *Proceedings of SIGCOMM 1988* (Cambridge, MA, Aug. 1988).

[12] CPLEX, I. I. 12.4, 2013.

[13] Dai, J., Huang, B., Li, L., and Harrison, L. Automatically partitioning packet processing applications for pipelined architectures. In *ACM SIGPLAN Notices* (2005), vol. 40, ACM, pp. 237–248.

[14] De Carli, L., Pan, Y., Kumar, A., Estan, C., and Sankaralingam, K. Plug: flexible lookup modules for rapid deployment of new protocols in high-speed routers. In *ACM SIGCOMM Computer Communication Review* (2009), vol. 39, ACM, pp. 207–218.

[15] Garey, M. R., Graham, R. L., Johnson, D. S., and Yao, A. C.-C. Resource constrained scheduling as generalized bin packing. *Journal of Combinatorial Theory, Series A 21*, 3 (1976), 257–298.

[16] Gibb, G., Varghese, G., Horowitz, M., and McKeown, N. Design principles for packet parsers. In *Architectures for Networking and Communications Systems (ANCS), 2013 ACM/IEEE Symposium on* (2013), IEEE, pp. 13–24.

[17] Intel Corporation. *Intel® Processors in Industrial Control and Automation Applications*. Intel White Paper, 2004.

[18] Jain, S., Kumar, A., Mandal, S., Ong, J., Poutievski, L., Singh, A., Venkata, S., Wanderer, J., Zhou, J., Zhu, M., et al. B4: Experience with a globally-deployed software defined WAN. In *Proceedings of the ACM SIGCOMM 2013 conference on SIGCOMM* (2013), ACM, pp. 3–14.

[19] Kozanitis, C., Huber, J., Singh, S., and Varghese, G. Leaping multiple headers in a single bound: wire-speed parsing using the kangaroo system. In *INFOCOM, 2010 Proceedings IEEE* (2010), IEEE, pp. 1–9.

[20] Lam, M., Sethi, R., Ullman, J., and Aho, A. Compilers: Principles, techniques, and tools, 2006.

[21] Mahalingam, D., Dutt, D., Duda, K., Agarwal, P., Kreeger, L., Sridhar, T., Bursell, M., and Wright, C. Vxlan: A framework for overlaying virtualized Layer 2 networks over Layer 3 networks. Internet-Draft draft-mahalingam-dutt-dcops-vxlan-00, IETF, 2011.

[22] Nowatzki, T., Sartin-Tarm, M., De Carli, L., Sankaralingam, K., Estan, C., and Robatmili, B. A general constraint-centric scheduling framework for spatial architectures. *ACM SIGPLAN Notices 48*, 6 (2013), 495–506.

[23] NVIDIA Corporation. *NVIDIA's Next Generation CUDA^{TM} Compute Architecture: Fermi^{TM}*. NVIDIA White Paper, 2009.

[24] Open Networking Foundation. *Software-Defined Networking: The new norm for networks*. Open Networking Foundation White Paper, 2012.

[25] Ozdag, R. Intel® Ethernet Switch FM6000 Series-Software Defined Networking. *Intel Corporation* (2012), 8.

[26] Rinta-Aho, T., Nikander, P., Sahasrabuddhe, S. D., and Kempf, J. Click-to-netfpga toolchain.

[27] Song, H. Protocol-oblivious forwarding: Unleash the power of SDN through a future-proof forwarding plane. In *Proceedings of the second ACM SIGCOMM workshop on Hot topics in software defined networking* (2013), ACM, pp. 127–132.

[28] Xilinx, Inc. *DSP: Designing for Optimal Results*. Xilinx, Inc., 2005.

The Design and Implementation of Open vSwitch

Ben Pfaff[*], *Justin Pettit*[*], *Teemu Koponen*[*], *Ethan J. Jackson*[*],
Andy Zhou[*], *Jarno Rajahalme*[*], *Jesse Gross*[*], *Alex Wang*[*],
Jonathan Stringer[*], *Pravin Shelar*[*], *Keith Amidon*[†], *Martín Casado*[*]
[*]VMware [†]Awake Networks

Operational Systems Track

Abstract

We describe the design and implementation of Open vSwitch, a multi-layer, open source virtual switch for all major hypervisor platforms. Open vSwitch was designed de novo for networking in virtual environments, resulting in major design departures from traditional software switching architectures. We detail the advanced flow classification and caching techniques that Open vSwitch uses to optimize its operations and conserve hypervisor resources. We evaluate Open vSwitch performance, drawing from our deployment experiences over the past seven years of using and improving Open vSwitch.

1 Introduction

Virtualization has changed the way we do computing over the past 15 years; for instance, many datacenters are entirely virtualized to provide quick provisioning, spillover to the cloud, and improved availability during periods of disaster recovery. While virtualization is still to reach all types of workloads, the number of virtual machines has already exceeded the number of servers and further virtualization shows no signs of stopping [1].

The rise of server virtualization has brought with it a fundamental shift in datacenter networking. A new network access layer has emerged in which most network ports are virtual, not physical [5] – and therefore, the first hop switch for workloads increasingly often resides within the hypervisor. In the early days, these hypervisor "vSwitches" were primarily concerned with providing basic network connectivity. In effect, they simply mimicked their ToR cousins by extending physical L2 networks to resident virtual machines. As virtualized workloads proliferated, limits of this approach became evident: reconfiguring and preparing a physical network for new workloads slows their provisioning, and coupling workloads with physical L2 segments severely limits their mobility and scalability to that of the underlying network.

These pressures resulted in the emergence of network virtualization [19]. In network virtualization, virtual switches become the primary provider of network services for VMs, leaving physical datacenter networks with transportation of IP tunneled packets between hypervisors. This approach allows the virtual networks to be decoupled from their underlying physical networks, and by leveraging the flexibility of general purpose processors, virtual switches can provide VMs, their tenants, and administrators with logical network abstractions, services and tools identical to dedicated physical networks.

Network virtualization demands a capable virtual switch – forwarding functionality must be wired on a per virtual port basis to match logical network abstractions configured by administrators. Implementation of these abstractions, across hypervisors, also greatly benefits from fine-grained centralized coordination. This approach starkly contrasts with early virtual switches for which a static, mostly hard-coded forwarding pipelines had been completely sufficient to provide virtual machines with L2 connectivity to physical networks.

It was this context: the increasing complexity of virtual networking, emergence of network virtualization, and limitations of existing virtual switches, that allowed Open vSwitch to quickly gain popularity. Today, on Linux, its original platform, Open vSwitch works with most hypervisors and container systems, including Xen, KVM, and Docker. Open vSwitch also works "out of the box" on the FreeBSD and NetBSD operating systems and ports to the VMware ESXi and Microsoft Hyper-V hypervisors are underway.

In this paper, we describe the design and implementation of Open vSwitch [26, 29]. The key elements of its design, revolve around the performance required by the production environments in which Open vSwitch is commonly deployed, and the programmability demanded by network virtualization. Unlike traditional network appliances, whether software or hardware, which achieve high performance through specialization, Open vSwitch, by contrast, is designed for flexibility and general-purpose usage. It must achieve high performance without the luxury of specialization, adapting to differences in platforms supported, all while sharing resources with the hypervisor and its workloads. Therefore, this paper foremost concerns this tension – how Open vSwitch obtains high performance without sacrificing generality.

The remainder of the paper is organized as follows. Section 2 provides further background about virtualized environments while Section 3 describes the basic design of Open vSwitch. Afterward, Sections 4, 5, and 6 describe how the Open vSwitch design optimizes for the requirements of virtualized environments through flow caching, how caching has wide-reaching implications for the entire design, including its packet classifier, and how Open vSwitch manages its flow caches. Section 7 then evaluates the performance of Open vSwitch through classification and caching micro-benchmarks but also provides a view of Open vSwitch performance in a multi-tenant datacenter. Before concluding, we discuss ongoing, future and related work in Section 8.

2 Design Constraints and Rationale

The operating environment of a virtual switch is drastically different from the environment of a traditional network appliance. Below we briefly discuss constraints and challenges stemming from these differences, both to reveal the rationale behind the design choices of Open vSwitch and highlight what makes it unique.

Resource sharing. The performance goals of traditional network appliances favor designs that use dedicated hardware resources to achieve line rate performance in *worst-case* conditions. With a virtual switch on the other hand, resource conservation is critical. Whether or not the switch can keep up with worst-case line rate is secondary to maximizing resources available for the primary function of a hypervisor: running user workloads. That is, compared to physical environments, networking in virtualized environments optimizes for the *common case* over the worst-case. This is not to say worst-case situations are not important because they do arise in practice. Port scans, peer-to-peer rendezvous servers, and network monitoring all generate unusual traffic patterns but must be supported gracefully. This principle led us, *e.g.,* toward heavy use of flow caching and other forms of caching, which in common cases (with high hit rates) reduce CPU usage and increase forwarding rates.

Placement. The placement of virtual switches at the edge of the network is a source of both simplifications and complications. Arguably, topological location as a leaf, as well as sharing fate with the hypervisor and VMs remove many standard networking problems. The placement complicates scaling, however. It's not uncommon for a single virtual switch to have thousands of virtual switches as its peers in a mesh of point-to-point IP tunnels between hypervisors. Virtual switches receive forwarding state updates as VMs boot, migrate, and shut down and while virtual switches have relatively few (by networking standards) physical network ports directly attached, changes in remote hypervisors may affect local state. Especially in larger deployments of thousands (or more) of hypervisors, the forwarding state may be in constant flux. The prime example of a design influenced by this principle discussed in this paper is the Open vSwitch classification algorithm, which is designed for $O(1)$ updates.

SDN, use cases, and ecosystem. Open vSwitch has three additional unique requirements that eventually caused its design to differ from the other virtual switches:

First, Open vSwitch has been an *OpenFlow switch* since its inception. It is deliberately not tied to a single-purpose, tightly vertically integrated network control stack, but instead is re-programmable through OpenFlow [27]. This constrasts with a *feature datapath* model of other virtual switches [24, 39]: similar to forwarding ASICs, their packet processing pipelines are fixed. Only configuration of prearranged features is possible. (The Hyper-V virtual switch [24] can be extended by adding binary modules, but ordinarily each module only adds another single-purpose feature to the datapath.)

The flexibility of OpenFlow was essential in the early days of SDN but it quickly became evident that advanced use cases, such as network virtualization, result in long packet processing pipelines, and thus higher classification load than traditionally seen in virtual switches. To prevent Open vSwitch from consuming more hypervisor resources than competitive virtual switches, it was forced to implement flow caching.

Third, unlike any other major virtual switch, Open vSwitch is open source and multi-platform. In contrast to closed source virtual switches which all operate in a single environment, Open vSwitch's environment is usually selected by a user who chooses an operating system distribution and hypervisor. This has forced the Open vSwitch design to be quite modular and portable.

3 Design

3.1 Overview

In Open vSwitch, two major components direct packet forwarding. The first, and larger, component is `ovs-vswitchd`, a userspace daemon that is essentially the same from one operating system and operating environment to another. The other major component, a

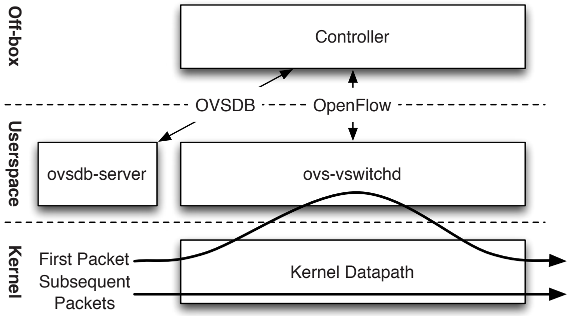


Figure 1: The components and interfaces of Open vSwitch. The first packet of a flow results in a miss, and the kernel module directs the packet to the userspace component, which caches the forwarding decision for subsequent packets into the kernel.

datapath kernel module, is usually written specially for the host operating system for performance.

Figure 1 depicts how the two main OVS components work together to forward packets. The datapath module in the kernel receives the packets first, from a physical NIC or a VM's virtual NIC. Either `ovs-vswitchd` has instructed the datapath how to handle packets of this type, or it has not. In the former case, the datapath module simply follows the instructions, called *actions*, given by `ovs-vswitchd`, which list physical ports or tunnels on which to transmit the packet. Actions may also specify packet modifications, packet sampling, or instructions to drop the packet. In the other case, where the datapath has not been told what to do with the packet, it delivers it to `ovs-vswitchd`. In userspace, `ovs-vswitchd` determines how the packet should be handled, then it passes the packet back to the datapath with the desired handling. Usually, `ovs-vswitchd` also tells the datapath to cache the actions, for handling similar future packets.

In Open vSwitch, flow caching has greatly evolved over time; the initial datapath was a *microflow cache*, essentially caching per transport connection forwarding decisions. In later versions, the datapath has two layers of caching: a microflow cache and a secondary layer, called a *megaflow cache*, which caches forwarding decisions for traffic aggregates beyond individual connections. We will return to the topic of caching in more detail in Section 4.

Open vSwitch is commonly used as an SDN switch, and the main way to control forwarding is OpenFlow [27]. Through a simple binary protocol, OpenFlow allows a controller to add, remove, update, monitor, and obtain statistics on flow tables and their flows, as well as to divert selected packets to the controller and to inject packets from the controller into the switch. In Open vSwitch, `ovs-vswitchd` receives OpenFlow flow tables from an SDN controller, matches any packets received from the datapath module against these OpenFlow tables, gathers the actions applied, and finally caches the result in the kernel datapath. This allows the datapath module to remain unaware of the particulars of the OpenFlow wire protocol, further simplifying it. From the OpenFlow controller's point of view, the caching and separation into user and kernel components are invisible implementation details: in the controller's view, each packet visits a series of OpenFlow flow tables and the switch finds the highest-priority flow whose conditions are satisfied by the packet, and executes its OpenFlow actions.

The flow programming model of Open vSwitch largely determines the use cases it can support and to this end, Open vSwitch has many extensions to standard OpenFlow to accommodate network virtualization. We will discuss these extensions shortly, but before that, we turn our focus on the performance critical aspects of this design: packet classification and the kernel-userspace interface.

3.2 Packet Classification

Algorithmic packet classification is expensive on general purpose processors, and packet classification in the context of OpenFlow is especially costly because of the generality of the form of the match, which may test any combination of Ethernet addresses, IPv4 and IPv6 addresses, TCP and UDP ports, and many other fields, including packet metadata such as the switch ingress port.

Open vSwitch uses a *tuple space search* classifier [34] for all of its packet classification, both kernel and userspace. To understand how tuple space search works, assume that all the flows in an Open vSwitch flow table matched on the same fields in the same way, *e.g.,* all flows match the source and destination Ethernet address but no other fields. A tuple search classifier implements such a flow table as a single hash table. If the controller then adds new flows with a different form of match, the classifier creates a second hash table that hashes on the fields matched in those flows. (The *tuple* of a hash table in a tuple space search classifier is, properly, the set of fields that form that hash table's key, but we often refer to the hash table itself as the tuple, as a kind of useful shorthand.) With two hash tables, a search must look in both hash tables. If there are no matches, the flow table doesn't contain a match; if there is a match in one hash table, that flow is the result; if there is a match in both, then the result is the flow with the higher priority. As the controller continues to add more flows with new forms of match, the classifier similarly expands to include a hash table for each unique match, and a search of the classifier must look in every hash table.

While the lookup complexity of tuple space search is far from the state of the art [8, 18, 38], it performs well with the flow tables we see in practice and has three attractive properties over decision tree classification algorithms. First, it supports efficient constant-time updates (an update translates to a single hash table operation), which makes it suitable for use with virtualized environments where a centralized controller may add and remove flows often, sometimes multiple times per second per hypervisor, in response to changes in the whole datacenter. Second, tuple space search generalizes to an arbitrary number of packet header fields, without any algorithmic change. Finally, tuple space search uses memory linear in the number of flows.

The relative cost of a packet classification is further amplified by the large number of flow tables that sophisticated SDN controllers use. For example, flow tables installed by the VMware network virtualization controller [19] use a minimum of about 15 table lookups per packet in its packet processing pipeline. Long pipelines are driven by two factors: reducing stages through cross-producting would often significantly increase the flow table sizes and developer preference to modularize the pipeline design. Thus, even more important than the performance of a single classifier lookup, it is to reduce the number of flow table lookups a single packet requires, on average.

3.3 OpenFlow as a Programming Model

Initially, Open vSwitch focused on a reactive flow programming model in which a controller responding to traffic installs microflows which match every supported OpenFlow field. This approach is easy to support for software switches and controllers alike, and early research suggested it was sufficient [3]. However, reactive programming of microflows soon proved impractical for use outside of small deployments and Open vSwitch had to adapt to proactive flow programming to limit its performance costs.

In OpenFlow 1.0, a microflow has about 275 bits of information, so that a flow table for every microflow would have 2^{275} or more entries. Thus, proactive population of flow tables requires support for wildcard matching to cover the header space of all possible packets. With a single table this results in a "cross-product problem": to vary the treatment of packets according to n_1 values of field A and n_2 values of field B, one must install $n_1 \times n_2$ flows in the general case, even if the actions to be taken based on A and B are independent. Open vSwitch soon introduced an extension action called *resubmit* that allows packets to consult multiple flow tables (or the same table multiple times), aggregating the resulting actions. This solves the cross-product problem, since one table can contain n_1 flows that consult A and another table n_2 flows that consult B. The resubmit action also enables a form of programming based on multiway branching based on the value of one or more fields. Later, OpenFlow vendors focusing on hardware sought a way to make better use of the multiple tables consulted in series by forwarding ASICs, and OpenFlow 1.1 introduced multi-table support. Open vSwitch adopted the new model but retained its support for the resubmit action for backward compatibility and because the new model did not allow for recursion but only forward progress through a fixed table pipeline.

At this point, a controller could implement programs in Open vSwitch flow tables that could make decisions based on packet headers using arbitrary chains of logic, but they had no access to temporary storage. To solve that problem, Open vSwitch extended OpenFlow in another way, by adding meta-data fields called "registers" that flow tables could match, plus additional actions to modify and copy them around. With this, for instance, flows could decide a physical destination early in the pipeline, then run the packet through packet processing steps identical regardless of the chosen destination, until sending the packet, possibly using destination-specific instructions. As another example, VMware's NVP network virtualization controller [19] uses registers to keep track of a packet's progress through a logical L2 and L3 topology implemented as "logical datapaths" that it overlays on the physical OpenFlow pipeline.

OpenFlow is specialized for flow-based control of a switch. It cannot create or destroy OpenFlow switches, add or remove ports, configure QoS queues, associate OpenFlow controller and switches, enable or disable STP (Spanning Tree Protocol), etc. In Open vSwitch, this functionality is controlled through a separate component, the *configuration database*. To access the configuration database, an SDN controller may connect to `ovsdb-server` over the OVSDB protocol [28], as shown in Figure 1. In general, in Open vSwitch, OpenFlow controls potentially fast-changing and ephemeral data such as the flow table, whereas the configuration database contains more durable state.

4 Flow Cache Design

This section describes the design of flow caching in Open vSwitch and how it evolved to its current state.

4.1 Microflow Caching

In 2007, when the development of the code that would become Open vSwitch started on Linux, only in-kernel packet forwarding could realistically achieve good performance, so the initial implementation put all OpenFlow processing into a kernel module. The module received a packet from a NIC or VM, classified through the OpenFlow table (with standard OpenFlow matches and actions), modified it as necessary, and finally sent it to another port. This approach soon became impractical because of the relative difficulty of developing in the kernel and distributing and updating kernel modules. It also became clear that an in-kernel OpenFlow implementation would not be acceptable as a contribution to upstream Linux, which is an important requirement for mainstream acceptance for software with kernel components.

Our solution was to reimplement the kernel module as a *microflow cache* in which a single cache entry exact matches with all the packet header fields supported by OpenFlow. This allowed radical simplification, by implementing the kernel module as a simple hash table rather than as a complicated, generic packet classifier, supporting arbitrary fields and masking. In this design, cache entries are extremely fine-grained and match *at most* packets of a single transport connection: even for a single transport connection, a change in network path and hence in IP TTL field would result in a miss, and would divert a packet to userspace, which consulted the actual OpenFlow flow table to decide how to forward it. This implies that the critical performance dimension is flow setup time, the time that it takes for the kernel to report a microflow "miss" to userspace and for userspace to reply.

Over multiple Open vSwitch versions, we adopted several techniques to reduce flow setup time with the microflow cache. Batching flow setups that arrive together improved flow setup performance about 24%, for example, by reducing the average number of system calls required to set up a given microflow. Eventually, we also distributed flow setup load over multiple userspace threads to benefit from multiple CPU cores. Drawing inspiration from CuckooSwitch [42], we adopted optimistic concurrent cuckoo hashing [6] and RCU [23] techniques to implement nonblocking multiple-reader, single-writer flow tables.

After general optimizations of this kind customer feedback drew us to focus on performance in latency-sensitive applications, and that required us to reconsider our simple caching design.

4.2 Megaflow Caching

While the microflow cache works well with most traffic patterns, it suffers serious performance degradation when faced with large numbers of short lived connections. In this case, many packets miss the cache, and must not only cross the kernel-userspace boundary, but also execute a long series of expensive packet classifications. While batching and multithreading can somewhat alleviate this stress, they are not sufficient to fully support this workload.

We replaced the microflow cache with a *megaflow cache*. The megaflow cache is a single flow lookup table that supports generic matching, *i.e.,* it supports caching forwarding decisions for larger aggregates of traffic than connections. While it more closely resembles a generic OpenFlow table than the microflow cache does, due to its support for arbitrary packet field matching, it is still strictly simpler and lighter in runtime for two primary reasons. First, it does not have priorities, which speeds up packet classification: the in-kernel tuple space search implementation can terminate as soon as it finds any match, instead of continuing to look for a higher-priority match until all the mask-specific hash tables are inspected. (To avoid ambiguity, userspace installs only disjoint megaflows, those whose matches do not overlap.) Second, there is only one megaflow classifier, instead of a pipeline of them, so userspace installs megaflow entries that collapse together the behavior of all relevant OpenFlow tables.

The cost of a megaflow lookup is close to the general-purpose packet classifier, even though it lacks support for flow priorities. Searching the megaflow classifier requires searching each of its hash tables until a match is found; and as discussed in Section 3.2, each unique kind of match in a flow table yields a hash table in the classifier. Assuming that each hash table is equally likely to contain a match, matching packets require searching $(n+1)/2$ tables on average, and non-matching packets require searching all n. Therefore, for $n > 1$, which is usually the case, a classifier-based megaflow search requires more hash table lookups than a microflow cache. Megaflows by themselves thus yield a trade-off: one must bet that the per-microflow benefit of avoiding an extra trip to userspace outweighs the per-packet cost of the extra hash lookups in form of megaflow lookup.

Open vSwitch addresses the costs of megaflows by retaining the microflow cache as a first-level cache, consulted before the megaflow cache. This cache is a hash table that maps from a microflow to its matching megaflow. Thus, after the first packet in a microflow passes through the kernel megaflow table, requiring a search of the kernel classifier, this exact-match cache allows subsequent packets in the same microflow to get quickly directed to the appropriate megaflow. This reduces the cost of megaflows from per-packet to per-microflow. The exact-match cache is a true cache in that its activity is not visible to userspace, other than through its effects on performance.

A megaflow flow table represents an active subset of the cross-product of all the userspace OpenFlow flow tables. To avoid the cost of proactive crossproduct computation and to populate the megaflow cache only with entries relevant for current forwarded traffic, the Open vSwitch userspace daemon computes the cache entries incrementally and reactively. As Open vSwitch processes a packet through userspace flow tables, classifying the packet at every table, it tracks the packet field bits that were consulted as part of the classification algorithm. The generated megaflow must match any field (or part of a field) whose value was used as part of the decision. For example, if the classifier looks at the IP destination field in any OpenFlow table as part of its pipeline, then the megaflow cache entry's condition must match on the destination IP as well. This means that incoming packets drive the cache population, and as the aggregates of the traffic evolve, new entries are populated and old entries removed.

The foregoing discussion glosses over some details. The basic algorithm, while correct, produces match conditions that are more specific than necessary, which translates to suboptimal cache hit rates. Section 5, below, describes how Open vSwitch modifies tuple space search to yield better megaflows for caching. Afterward, Section 6 addresses cache invalidation.

5 Caching-aware Packet Classification

We now turn our focus on the refinements and improvements we made to the basic tuple search algorithm (summarized in Section 3.2) to improve its suitability for flow caching.

5.1 Problem

As Open vSwitch userspace processes a packet through its OpenFlow tables, it tracks the packet field bits that were consulted as part of the forwarding decision. This bitwise tracking of packet header fields is very effective in constructing the megaflow entries with simple OpenFlow flow tables.

For example, if the OpenFlow table only looks at Ethernet addresses (as would a flow table based on L2 MAC learning), then the megaflows it generates will also look only at Ethernet addresses. For example, port scans (which do not vary Ethernet addresses) will not cause packets to go to userspace as their L3 and L4 header fields will be wildcarded resulting in near-ideal megaflow cache hit rates. On the other hand, if even one flow entry in the table matches on the TCP destination port, tuple space search will consider the TCP destination port of every packet. Then every megaflow will also match on the TCP destination port, and port scan performance again drops.

We do not know of an efficient online algorithm to generate optimal, least specific megaflows, so in development we have focused our attention on generating increasingly good approximations. Failing to match a field that must be included can cause incorrect packet forwarding, which makes such errors unacceptable, so our approximations are biased toward matching on more fields than necessary. The following sections describe improvements of this type that we have integrated into Open vSwitch.

```
function PRIORITYSORTEDTUPLESEARCH($H$)
    $B \leftarrow$ NULL        /* Best flow match so far. */
    for tuple $T$ in descending order of $T.pri\_max$ do
        if $B \neq$ NULL and $B.pri \geq T.pri\_max$ then
            return $B$
        if $T$ contains a flow $F$ matching $H$ then
            if $B =$ NULL or $F.pri > B.pri$ then
                $B \leftarrow F$
    return $B$
```

Figure 2: Tuple space search for target packet headers H, with priority sorting.

5.2 Tuple Priority Sorting

Lookup in a tuple space search classifier ordinarily requires searching every tuple. Even if a search of an early tuple finds a match, the search must still look in the other tuples because one of them might contain a matching flow with a higher priority.

We improved on this by tracking, in each tuple T, the maximum priority $T.pri\_max$ of any flow entry in T. We modified the lookup code to search tuples from greatest to least maximum priority, so that a search that finds a matching flow F with priority $F.pri$ can terminate as soon as it arrives at a tuple whose maximum priority is $F.pri$ or less, since at that point no better match can be found. Figure 2 shows the algorithm in detail.

As an example, we examined the OpenFlow table installed by a production deployment of VMware's NVP controller [19]. This table contained 29 tuples. Of those 29 tuples, 26 contained flows of a single priority, which makes intuitive sense because flows matching a single tuple tend to share a purpose and therefore a priority. When searching in descending priority order, one can always terminate immediately following a successful match in such a tuple. Considering the other tuples, two contained flows with two unique priorities that were higher than those in any subsequent tuple, so any match in either of these tuples terminated the search. The final tuple contained flows with five unique priorities ranging from 32767 to 36866; in the worst case, if the lowest priority flows matched in this tuple, then the remaining tuples with $T.pri\_max > 32767$ (up to 20 tuples based on this tuple's location in the sorted list), must also be searched.

5.3 Staged Lookup

Tuple space search searches each tuple with a hash table lookup. In our algorithm to construct the megaflow matching condition, this hash table lookup means that the megaflow must match all the bits of fields included in the tuple, even if the tuple search fails, because every one of those fields and their bits may have affected the lookup result so far. When the tuple matches on a field that varies often from flow to flow, *e.g.,* the TCP source port, the generated megaflow is not much more useful than installing a microflow would be because it will only match a single TCP stream.

This points to an opportunity for improvement. If one could search a tuple on a subset of its fields, and determine with this search that the tuple could not possibly match, then the generated megaflow would only need to match on the subset of fields, rather than all the fields in the tuple.

The tuple implementation as a hash table over all its fields made such an optimization difficult. One cannot search a hash table on a subset of its key. We considered other data structures. A trie would allow a search on any prefix of fields, but it would also increase the number of memory accesses required by a successful search from $O(1)$ to $O(n)$ in the length of the tuple fields. Individual per-field hash tables had the same drawback. We did not consider data structures larger than $O(n)$ in the number of flows in a tuple, because OpenFlow tables can have hundreds of thousands of flows.

The solution we implemented statically divides fields into four groups, in decreasing order of traffic granularity: metadata (*e.g.,* the switch ingress port), L2, L3, and L4. We changed each tuple from a single hash table to an array of four hash tables, called *stages*: one over metadata fields only, one over metadata and L2 fields, one over metadata, L2, and L3 fields, and one over all fields. (The latter is the same as the single hash table in the previous implementation.) A lookup in a tuple searches each of its stages in order. If any search turns up no match, then the overall search of the tuple also fails, and only the fields included in the stage last searched must be added to the megaflow match.

This optimization technique would apply to any subsets of the supported fields, not just the layer-based subsets we used. We divided fields by protocol layer because, as a rule of thumb, in TCP/IP, inner layer headers tend to be more diverse than outer layer headers. At L4, for example, the TCP source and destination ports change on a per-connection basis, but in the metadata layer only a relatively small and static number of ingress ports exist.

Each stage in a tuple includes all of the fields in earlier stages. We chose this arrangement, although the technique does not require it, because then hashes could be computed incrementally from one stage to the next, and profiling had shown hash computation to be a significant cost (with or without staging).

With four stages, one might expect the time to search a tuple to quadruple. Our measurements show that, in fact, classification speed actually improves slightly in practice because, when a search terminates at any early stage, the classifier does not have to compute the full hash of all the fields covered by the tuple.

This optimization fixes a performance problem observed in production deployments. The NVP controller uses Open vSwitch to implement multiple isolated logical datapaths (further interconnected to form logical networks). Each logical datapath is independently configured. Suppose that some logical datapaths are configured with ACLs that allow or deny traffic based on L4 (*e.g.,* TCP or UDP) port numbers. Megaflows for traffic on these logical datapaths must match on the L4 port to enforce the ACLs. Megaflows for traffic on other logical datapaths need not and, for performance, should not match on L4 port. Before this optimization, however, all generated megaflows matched on L4 port because a classifier search had to pass through a tuple that matched on L4 port. The optimization allows megaflows for traffic on logical datapaths without L4 ACLs to avoid matching on L4 port, because the first three (or fewer) stages are enough to determine that there is no match.

5.4 Prefix Tracking

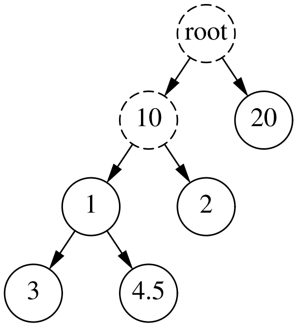
Flows in OpenFlow often match IPv4 and IPv6 subnets to implement routing. When all the flows that match on such a field use the same subnet size, *e.g.,* all match /16 subnets, this works out fine for constructing megaflows. If, on the other hand, different flows match different subnet sizes, like any standard IP routing table does, the constructed megaflows match the longest subnet prefix, *e.g.,* any host route (/32) forces all the megaflows to match full addresses. Suppose, for example, Open vSwitch is constructing a megaflow for a packet addressed to 10.5.6.7. If flows match subnet 10/8 and host 10.1.2.3/32, one could safely install a megaflow for 10.5/16 (because 10.5/16 is completely inside 10/8 and does not include 10.1.2.3), but without additional optimization Open vSwitch installs 10.5.6.7/32. (Our examples use only octet prefixes, *e.g.,* /8, /16, /24, /32, for clarity, but the implementation and the pseudocode shown later work in terms of bit prefixes.)

We implemented optimization of prefixes for IPv4 and IPv6 fields using a trie structure. If a flow table matches over an IP address, the classifier executes an LPM lookup for any such field *before* the tuple space search, both to determine the maximum megaflow prefix length required, as well as to determine which tuples can be skipped entirely without affecting correctness.[1] As an example, suppose an OpenFlow table contained flows that matched on some IPv4 field, as shown:

[1] This is a slight simplification for improved clarity; the actual implementation reverts to prefix tracking if staged lookups have concluded to include an IP field to the match.

20 /8
10.1 /16
10.2 /16
10.1.3 /24
10.1.4.5/32

These flows correspond to the following trie, in which a solid circle represents one of the address matches listed above and a dashed circle indicates a node that is present only for its children:



To determine the bits to match, Open vSwitch traverses the trie from the root down through nodes with labels matching the corresponding bits in the packet's IP address. If traversal reaches a leaf node, then the megaflow need not match the remainder of the address bits, *e.g.,* in our example 10.1.3.5 would be installed as 10.1.3/24 and 20.0.5.1 as 20/8. If, on the other hand, traversal stops due to the bits in the address not matching any of the corresponding labels in the tree, the megaflow must be constructed to match up to and including the bits that could not be found, *e.g.,* 10.3.5.1 must be installed as 10.3/16 and 30.10.5.2 as 30/8.

The trie search result also allows Open vSwitch to skip searching some tuples. Consider the address 10.1.6.1. A search of the above trie for this address terminates at the node labeled 1, failing to find a node to follow for the address's third octet. This means that no flow in the flow table with an IP address match longer than 16 bits matches the packet, so the classifier lookup can skip searching tuples for the flows listed above with /24 and /32 prefixes.

Figure 3 gives detailed pseudocode for the prefix matching algorithm. Each node is assumed to have members *bits*, the bits in the particular node (at least one bit, except that the root node may be empty); *left* and *right*, the node's children (or NULL); and *n_rules*, the number of rules in the node (zero if the node is present only for its children, otherwise nonzero). It returns the number of bits that must be matched, allowing megaflows to be improved, and a bit-array in which 0-bits designate matching lengths for tuples that Open vSwitch may skip searching, as described above.

```
function TRIESEARCH($value$, $root$)
    $node \leftarrow root, prev \leftarrow$ NULL
    $plens \leftarrow$ bit-array of len($value$) 0-bits
    $i \leftarrow 0$
    while $node \neq$ NULL do
        $c \leftarrow 0$
        while $c < \text{len}(node.bits)$ do
            if $value[i] \neq node.bits[c]$ then
                return $(i+1, plens)$
            $c \leftarrow c+1, i \leftarrow i+1$
        if $node.n\_rules > 0$ then
            $plens[i-1] \leftarrow 1$
        if $i \geq \text{len}(value)$ then
            return $(i, plens)$
        $prev \leftarrow node$
        if $value[i] = 0$ then
            $node \leftarrow node.left$
        else
            $node \leftarrow node.right$
    if $prev \neq$ NULL and $prev$ has at least one child then
        $i \leftarrow i+1$
    return $(i, plens)$
```

Figure 3: Prefix tracking pseudocode. The function searches for *value* (*e.g.,* an IP address) in the trie rooted at node *root*. It returns the number of bits at the beginning of *value* that must be examined to render its matching node unique, and a bit-array of possible matching lengths. In the pseudocode, $x[i]$ is bit i in x and len(x) the number of bits in x.

While this algorithm optimizes longest-prefix match lookups, it improves megaflows even when no flow explicitly matches against an IP prefix. To implement a longest prefix match in OpenFlow, the flows with longer prefix must have higher priorities, which will allow the tuple priority sorting optimization in Section 5.2 to skip prefix matching tables after the longest match is found, but this alone causes megaflows to unwildcard address bits according to the longest prefix in the table. The main practical benefit of this algorithm, then, is to prevent policies (such as a high priority ACL) that are applied to a specific host from forcing all megaflows to match on a full IP address. This algorithm allows the megaflow entries only to match with the high order bits sufficient to differentiate the traffic from the host with ACLs.

We also eventually adopted prefix tracking for L4 transport port numbers. Similar to IP ACLs, this prevents high-priority ACLs that match specific transport ports (*e.g.,* to block SMTP) from forcing all megaflows to match the entire transport port fields, which would again reduce the megaflow cache to a microflow cache [32].

5.5 Classifier Partitioning

The number of tuple space searches can be further reduced by skipping tuples that cannot possibly match. OpenFlow supports setting and matching metadata fields during a packet's trip through the classifier. Open vSwitch partitions the classifier based on a particular metadata field. If the current value in that field does not match any value in a particular tuple, the tuple is skipped altogether.

While Open vSwitch does not have a fixed pipeline like traditional switches, NVP often configures each lookup in the classifier as a stage in a pipeline. These stages match on a fixed number of fields, similar to a tuple. By storing a numeric indicator of the pipeline stage into a specialized metadata field, NVP provides a hint to the classifier to efficiently only look at pertinent tuples.

6 Cache Invalidation

The flip side of caching is the complexity of managing the cache. In Open vSwitch, the cache may require updating for a number of reasons. Most obviously, the controller can change the OpenFlow flow table. OpenFlow also specifies changes that the switch should take on its own in reaction to various events, *e.g.,* OpenFlow "group" behavior can depend on whether carrier is detected on a network interface. Reconfiguration that turns features on or off, adds or removes ports, etc., can affect packet handling. Protocols for connectivity detection, such as CFM [10] or BFD [14], or for loop detection and avoidance, *e.g.,* (Rapid) Spanning Tree Protocol, can influence behavior. Finally, some OpenFlow actions and Open vSwitch extensions change behavior based on network state, *e.g.,* based on MAC learning.

Ideally, Open vSwitch could precisely identify the megaflows that need to change in response to some event. For some kinds of events, this is straightforward. For example, when the Open vSwitch implementation of MAC learning detects that a MAC address has moved from one port to another, the datapath flows that used that MAC are the ones that need an update. But the generality of the OpenFlow model makes precise identification difficult in other cases. One example is adding a new flow to an OpenFlow table. Any megaflow that matched a flow in that OpenFlow table whose priority is less than the new flow's priority should potentially now exhibit different behavior, but we do not know how to efficiently (in time and space) identify precisely those flows.[2] The problem is worsened further by long sequences of OpenFlow flow table lookups. We concluded that precision is not practical in the general case.

Therefore, early versions of Open vSwitch divided changes that could require the behavior of datapath flows to change into two groups. For the first group, the changes whose effects were too broad to precisely identify the needed changes, Open vSwitch had to examine every datapath flow for possible changes. Each flow had to be passed through the OpenFlow flow table in the same way as it was originally constructed, then the generated actions compared against the ones currently installed in the datapath. This can be time-consuming if there are many datapath flows, but we have not observed this to be a problem in practice, perhaps because there are only large numbers of datapath flows when the system actually has a high network load, making it reasonable to use more CPU on networking. The real problem was that, because Open vSwitch was single-threaded, the time spent re-examining all of the datapath flows blocked setting up new flows for arriving packets that did not match any existing datapath flow. This added high latency to flow setup for those packets, greatly increased the overall variability of flow setup latency, and limited the overall flow setup rate. Through version 2.0, therefore, Open vSwitch limited the maximum number of cached flows installed in the datapath to about 1,000, increased to 2,500 following some optimizations, to minimize these problems.

The second group consisted of changes whose effects on datapath flows could be narrowed down, such as MAC learning table changes. Early versions of Open vSwitch implemented these in an optimized way using a technique called *tags*. Each property that, if changed, could require megaflow updates was given one of these tags. Also, each megaflow was associated with the tags for all of the properties on which its actions depended, *e.g.,* if the actions output the packet to port *x* because the packet's destination MAC was learned to be on that port, then the megaflow is associated with the tag for that learned fact. Later, if that MAC learned port changed, Open vSwitch added the tag to a set of tags that accumulated changes. In batches, Open vSwitch scanned the megaflow table for megaflows that had at least one of the changed tags, and checked whether their actions needed an update.

Over time, as controllers grew more sophisticated and flow tables more complicated, and as Open vSwitch added more actions whose behavior changed based on network state, each datapath flow became marked with more and more tags. We had implemented tags as Bloom filters [2], which meant that each additional tag caused more "false positives" for revalidation, so now most or all flows required examination whenever any state changed. By Open vSwitch version 2.0, the effectiveness of tags had declined so much that to simplify the code Open vSwitch abandoned them altogether in favor of always revalidating the entire datapath flow table.

Since tags had been one of the ways we sought to minimize flow setup latency, we now looked for other ways. In Open vSwitch 2.0, toward that purpose, we divided userspace into multiple threads. We broke flow setup into separate threads so that it did not have to wait behind

[2] Header space analysis [16] provides the algebra to identify the flows but the feasibility of efficient, online analysis (such as in [15]) in this context remains an open question.

revalidation. Datapath flow eviction, however, remained part of the single main thread and could not keep up with multiple threads setting up flows. Under heavy flow setup load, though, the rate at which eviction can occur is critical, because userspace must be able to delete flows from the datapath as quickly as it can install new flows, or the datapath cache will quickly fill up. Therefore, in Open vSwitch 2.1 we introduced multiple dedicated threads for cache revalidation, which allowed us to scale up the revalidation performance to match the flow setup performance and to greatly increase the kernel cache maximum size, to about 200,000 entries. The actual maximum is dynamically adjusted to ensure that total revalidation time stays under 1 second, to bound the amount of time that a stale entry can stay in the cache.

Open vSwitch userspace obtains datapath cache statistics by periodically (about once per second) polling the kernel module for every flow's packet and byte counters. The core use of datapath flow statistics is to determine which datapath flows are useful and should remain installed in the kernel and which ones are not processing a significant number of packets and should be evicted. Short of the table's maximum size, flows remain in the datapath until they have been idle for a configurable amount of time, which now defaults to 10 s. (Above the maximum size, Open vSwitch drops this idle time to force the table to shrink.) The threads that periodically poll the kernel for per flow statistics also use those statistics to implement OpenFlow's per-flow packet and byte count statistics and flow idle timeout features. This means that OpenFlow statistics are themselves only periodically updated.

The above describes how userspace invalidates the datapath's megaflow cache. Maintenance of the first-level microflow cache (discussed in Section 4) is much simpler. A microflow cache entry is only a hint to the first hash table to search in the general tuple space search. Therefore, a stale microflow cache entry is detected and corrected the first time a packet matches it. The microflow cache has a fixed maximum size, with new microflows replacing old ones, so there is no need to periodically flush old entries. We use a pseudo-random replacement policy, for simplicity, and have found it to be effective in practice.

7 Evaluation

The following sections examine Open vSwitch performance in production and in microbenchmarks.

7.1 Performance in Production

We examined 24 hours of Open vSwitch performance data from the hypervisors in a large, commercial multi-tenant data center operated by Rackspace. Our data set contains statistics polled every 10 minutes from over 1,000 hypervisors running Open vSwitch to serve mixed tenant workloads in network virtualization setting.

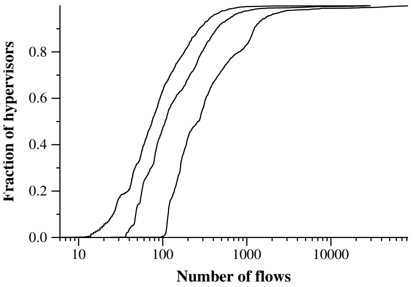


Figure 4: Min/mean/max megaflow flow counts observed.

Cache sizes. The number of active megaflows gives us an indication about practical megaflow cache sizes Open vSwitch handles. In Figure 4, we show the CDF for minimum, mean and maximum counts during the observation period. The plots show that small megaflow caches are sufficient in practice: 50% of the hypervisors had mean flow counts of 107 or less. The 99th percentile of the maximum flows was still just 7,033 flows. For the hypervisors in this environment, Open vSwitch userspace can maintain a sufficiently large kernel cache. (With the latest Open vSwitch mainstream version, the kernel flow limit is set to 200,000 entries.)

Cache hit rates. Figure 5 shows the effectiveness of caching. The solid line plots the overall cache hit rate across each of the 10-minute measurement intervals across the entire population of hypervisors. The overall cache hit rate was 97.7%. The dotted line includes just the 25% of the measurement periods in which the fewest packets were forwarded, in which the caching was less effective than overall, achieving a 74.7% hit rate. Intuitively, caching is less effective (and unimportant) when there is little to cache. Open vSwitch caching is most effective when it is most useful: when there is a great deal of traffic to cache. The dashed line, which includes just the 25% of the measurement periods in which the most packets were forwarded, demonstrates this: during these periods, the hit rate rises slightly above the overall average to 98.0%.

The vast majority of the hypervisors in this data center do not experience high volume traffic from their workloads. Figure 6 depicts this: 99% of the hypervisors see fewer than 79,000 packets/s to hit their caches (and fewer than 1500 flow setups/s to enter userspace due to misses).

CPU usage. Our statistics gathering process cannot separate Open vSwitch kernel load from the rest of the kernel load, so we focus on Open vSwitch userspace. As we

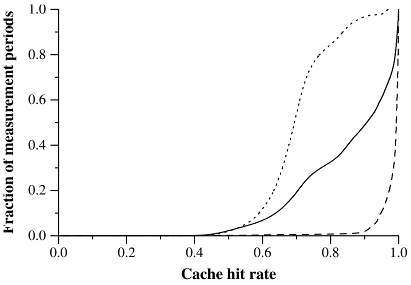


Figure 5: Hit rates during all (solid), busiest (dashed), and slowest (dotted) periods.

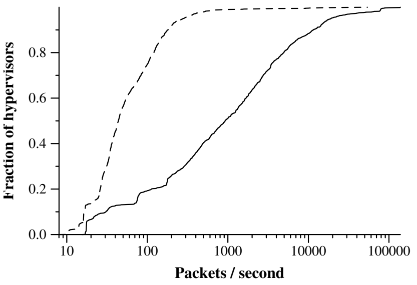


Figure 6: Cache hit (solid) and miss (dashed) packet counts.

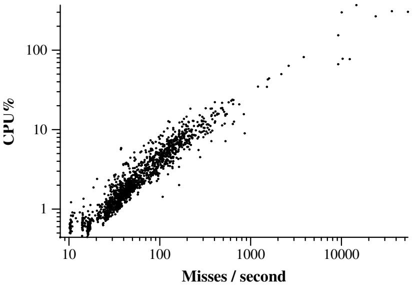


Figure 7: Userspace daemon CPU load as a function of misses/s entering userspace.

will show in Section 7.2, the megaflow CPU usage itself is in line with Linux bridging and less of a concern. In Open vSwitch, the userspace load is largely due to the misses in kernel and Figure 7 depicts this. (Userspace CPU load can exceed 100% due to multithreading.) We observe that 80% of the hypervisors averaged 5% CPU or less on `ovs-vswitchd`, which has been our traditional goal. Over 50% of hypervisors used 2% CPU or less.

Outliers. The upper right corner of Figure 7 depicts a number of hypervisors using large amounts of CPU to process many misses in userspace. We individually examined the six most extreme cases, where Open vSwitch averaged over 100% CPU over the 24 hour period. We found that all of these hypervisors exhibited a previously unknown bug in the implementation of prefix tracking, such that flows that match on an ICMP type or code caused all TCP flows to match on the entire TCP source or destination port, respectively. We believe we have fixed this bug in Open vSwitch 2.3, but the data center was not upgraded in time to verify in production.

7.2 Caching Microbenchmarks

We ran microbenchmarks with a simple flow table designed to compactly demonstrate the benefits of the caching-aware packet classification algorithm. We used the following OpenFlow flows, from highest to lowest priority. We omit the actions because they are not significant for the discussion:

arp (1)
ip ip_dst=11.1.1.1/16 (2)
tcp ip_dst=9.1.1.1 tcp_src=10 tcp_dst=10 (3)
ip ip_dst=9.1.1.1/24 (4)

With this table, with no caching-aware packet classification, any TCP packet will always generate a megaflow that matches on TCP source and destination ports, because flow #3 matches on those fields. With priority sorting (Section 5.2), packets that match flow #2 can omit matching on TCP ports, because flow #3 is never considered. With staged lookup (Section 5.3), IP packets not destined to 9.1.1.1 never need to match on TCP ports, because flow #3 is identified as non-matching after considering only the IP destination address. Finally, address prefix tracking (Section 5.4) allows megaflows to ignore some of the bits in IP destination addresses even though flow #3 matches on the entire address.

Optimizations	ktps	Flows	Masks	CPU%
Megaflows disabled	37	1,051,884	1	45/ 40
No optimizations	56	905,758	3	37/ 40
Priority sorting only	57	794,124	4	39/ 45
Prefix tracking only	95	13	10	0/ 15
Staged lookup only	115	14	13	0/ 15
All optimizations	117	15	14	0/ 20

Table 1: Performance testing results for classifier optimizations. Each row reports the measured number of Netperf `TCP_CRR` transactions per second, in thousands, along with the number of kernel flows, kernel masks, and user and kernel CPU usage.

Microflows	Optimizations	ktps	Tuples/pkt	CPU%
Enabled	Enabled	120	1.68	0/ 20
Disabled	Enabled	92	3.21	0/ 18
Enabled	Disabled	56	1.29	38/ 40
Disabled	Disabled	56	2.45	40/ 42

Table 2: Effects of microflow cache. Each row reports the measured number of Netperf `TCP_CRR` transactions per second, in thousands, along with the average number of tuples searched by each packet and user and kernel CPU usage.

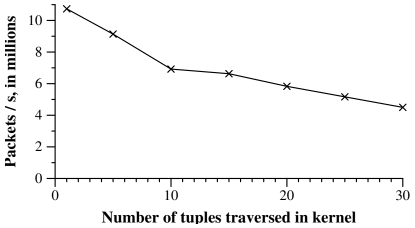


Figure 8: Forwarding rate in terms of the average number of megaflow tuples searched, with the microflow cache disabled.

Cache layer performance. We measured first the baseline performance of each Open vSwitch cache layer. In all following tests, Open vSwitch ran on a Linux server with two 8-core, 2.0 GHz Xeon processors and two Intel 10-Gb NICs. To generate many connections, we used Netperf's TCP_CRR test [25], which repeatedly establishes a TCP connection, sends and receives one byte of traffic, and disconnects. The results are reported in transactions per second (tps). Netperf only makes one connection attempt at a time, so we ran 400 Netperf sessions in parallel and reported the sum.

To measure the performance of packet processing in Open vSwitch userspace, we configured ovs-vswitchd to disable megaflow caching, by setting up only microflow entries in the datapath. As shown in Table 1, this yielded 37 ktps in the TCP_CRR test, with over one million kernel flow entries, and used about 1 core of CPU time.

To quantify the throughput of the megaflow cache by itself, we re-enabled megaflow caching, then disabled the kernel's microflow cache. Table 2 shows that disabling the microflow cache reduces TCP_CRR performance from 120 to 92 ktps when classifier optimizations are enabled. (When classifier optimizations are disabled, disabling the microflow cache has little effect because it is overshadowed by the increased number of trips to userspace.)

Figure 8 plots packet forwarding performance for long-lived flows as a function of the average number of tuples searched, with the kernel microflow cache disabled. In the same scenarios, with the microflow cache enabled, we measured packet forwarding performance of long-lived flows to be approximately 10.6 Mpps, independent of the number of tuples in the kernel classifier. Even searching only 5 tuples on average, the microflow cache improves performance by 1.5 Mpps, clearly demonstrating its value. To put these numbers in perspective in terms of raw hash lookup performance, we benchmarked our tuple space classifier in isolation: with a randomly generated table of half a million flow entries, the implementation is able to do roughly 6.8M hash lookups/s, on a single core – which translates to 680,000 classifications per second with 10 tuples.

Classifier optimization benefit. We measured the benefit of our classifier optimizations. Table 1 shows the improvement from individual optimizations and all of the optimizations together. Each optimization reduces the number of kernel flows needed to run the test. Each kernel flow corresponds to one trip between the kernel and userspace, so each reduction in flows also reduces userspace CPU time used. As can be seen from the table, as the number of kernel flows (Flows) declines, the number of tuples in the kernel flow table (Masks) increases, increasing the cost of kernel classification, but the measured reduction in kernel CPU time and increase in TCP_CRR shows that this is more than offset by the microflow cache and by fewer trips to userspace. The TCP_CRR test is highly sensitive to latency, demonstrating that latency decreases as well.

Comparison to in-kernel switch. We compared Open vSwitch to the Linux bridge, an Ethernet switch implemented entirely inside the Linux kernel. In the simplest configuration, the two switches achieved identical throughput (18.8 Gbps) and similar TCP_CRR connection rates (696 ktps for Open vSwitch, 688 for the Linux bridge), although Open vSwitch used more CPU (161% vs. 48%). However, when we added one flow to Open vSwitch to drop STP BPDU packets and a similar iptables rule to the Linux bridge, Open vSwitch performance and CPU usage remained constant whereas the Linux bridge connection rate dropped to 512 ktps and its CPU usage increased over 26-fold to 1,279%. This is because the built-in kernel functions have per-packet overhead, whereas Open vSwitch's overhead is generally fixed per-megaflow. We expect enabling other features, such as routing and a firewall, would similarly add CPU load.

8 Ongoing, Future, and Related Work

We now briefly discuss our current and planned efforts to improve Open vSwitch, and briefly cover related work.

8.1 Stateful Packet Processing

OpenFlow does not accommodate stateful packet operations, and thus, per-connection or per-packet forwarding state requires the controller to become involved. For this purpose, Open vSwitch allows running on-hypervisor "local controllers" in addition to a remote, primary controller. Because a local controller is an arbitrary program, it can maintain any amount of state across the packets that Open vSwitch sends it.

NVP includes, for example, a local controller that implements a stateful L3 daemon responsible for sending and processing ARPs. The L3 daemon populates an L3 ARP cache into a dedicated OpenFlow table (not managed by the primary controller) for quick forwarding of common case (packets with a known IP to MAC binding). The L3 daemon only receives packets resulting in an ARP cache miss and emits any necessary ARP requests to remote L3 daemons based on the packets received from Open vSwitch. While the connectivity between the local controller and Open vSwitch is local, the performance overhead is significant: a received packet traverses first from kernel to userspace daemon from which it traverses across a local socket (again via kernel) to a separate process.

For performance critical stateful packet operations, Open vSwitch relies on kernel networking facilities. For instance, a solid IP tunneling implementation requires (stateful) IP reassembly support. In a similar manner, transport connection tracking is a first practical requirement after basic L2/L3 networking; even most basic firewall security policies call for stateful filtering. OpenFlow is flexible enough to implement *static* ACLs but not stateful ones. For this, there's an ongoing effort to provide a new OpenFlow action that invokes a kernel module that provides metadata which the subsequent OpenFlow tables may use the connection state (new, established, related) in their forwarding decision. This "connection tracking" is the same technique used in many dedicated firewall appliances. Transitioning between kernel networking stack and kernel datapath module incurs overhead but avoids the duplication of functionality, critical in upstreaming kernel changes.

8.2 Userspace Networking

Improving the virtual switch performance through userspace networking is a timely topic due to NFV [9, 22]. In this model, packets are passed directly from the NIC to VM with minimal intervention by the hypervisor userspace/kernel, typically through shared memory between NIC, virtual switch, and VMs. To this end, there is an ongoing effort to add both DPDK [11] and netmap [30] support to Open vSwitch. Early tests indicate the Open vSwitch caching architecture in this context is similarly beneficial to kernel flow cache.

An alternative to DPDK that some in the Linux community are investigating is to reduce the overhead of going through the kernel. In particular, the SKB structure that stores packets in the Linux kernel is several cache lines large, contrary to the compact representation in DPDK and netmap. We expect the Linux community will make significant improvements in this regard.

8.3 Hardware Offloading

Over time, NICs have added hardware offloads for commonly needed functions that use excessive host CPU time. Some of these features, such as TCP checksum and segmentation offload, have proven very effective over time. Open vSwitch takes advantage of these offloads, and most others, which are just as relevant to virtualized environments. Specialized hardware offloads for virtualized environments have proven more elusive, though.

Offloading virtual switching entirely to hardware is a recurring theme (see, *e.g.,* [12]). This yields high performance, but at the cost of flexibility: a simple fixed function hardware switch effectively replaces the software virtual switch with no ability for the hypervisor to extend its functionality. The offload approach we currently find most promising is to enable NICs to accelerate kernel flow classification. The Flow Director feature on some Intel NICs has already been shown to be useful for classifying packets to separate queues [36]. Enhancing this feature simply to report the matching rule, instead of selecting the queue, would make it useful as such for megaflow classification. Even if the TCAM size were limited, or if the TCAM did not support all the fields that the datapath uses, it could speed up software classification by reducing the number of hash table searches – without limiting the flexibility since the actions would still take place in the host CPU.

8.4 Related Work

Flow caching. The benefits of flow caching generally have been argued by many in the community [4, 13, 17, 31, 41]. Lee et al. [21] describes how to augment the limited capacity of a hardware switch's flow table using a software flow cache, but does not mention problems with flows of different forms or priorities. CacheFlow [13], like Open vSwitch, caches a set of OpenFlow flows in a fast path, but CacheFlow requires the fast path to directly implement all the OpenFlow actions and requires building a full flow dependency graph in advance.

Packet classification. Classification is a well-studied problem [37]. Many classification algorithms only work with static sets of flows, or have expensive incremental update procedures, making them unsuitable for dynamic OpenFlow flow tables [7, 8, 33, 38, 40]. Some classifiers require memory that is quadratic or exponential in the number of flows [8, 20, 35]. Other classifiers work only with 2 to 5 fields [35], whereas OpenFlow 1.0 has 12 fields and later versions have more. (The effective number of fields is much higher with classifiers that must treat each bit of a bitwise matchable field as an individual field.)

9 Conclusion

We described the design and implementation of Open vSwitch, an open source, multi-platform OpenFlow virtual switch. Open vSwitch has simple origins but its performance has been gradually optimized to match the requirements of multi-tenant datacenter workloads, which has necessitated a more complex design. Given its operating environment, we anticipate no change of course but expect its design only to become more distinct from traditional network appliances over time.

References

[1] T. J. Bittman, G. J. Weiss, M. A. Margevicius, and P. Dawson. Magic Quadrant for x86 Server Virtualization Infrastructure. Gartner, June 2013.

[2] B. H. Bloom. Space/Time Trade-offs in Hash Coding with Allowable Errors. *Commun. ACM*, 13(7):422–426, July 1970.

[3] M. Casado, M. J. Freedman, J. Pettit, J. Luo, N. McKeown, and S. Shenker. Ethane: Taking Control of the Enterprise. In *Proc. of SIGCOMM*, 2007.

[4] M. Casado, T. Koponen, D. Moon, and S. Shenker. Rethinking Packet Forwarding Hardware. In *Proc. of HotNets*, 2008.

[5] Crehan Research Inc. and VMware Estimate, Mar. 2013.

[6] B. Fan, D. G. Andersen, and M. Kaminsky. Memc3: Compact and concurrent memcache with dumber caching and smarter hashing. In *NSDI*, volume 13, pages 385–398, 2013.

[7] A. Feldman and S. Muthukrishnan. Tradeoffs for Packet classification. In *Proc. of INFOCOM*, volume 3, pages 1193–1202 vol.3, Mar 2000.

[8] P. Gupta and N. McKeown. Packet Classification Using Hierarchical Intelligent Cuttings. In *Hot Interconnects VII*, pages 34–41, 1999.

[9] J. Hwang, K. K. Ramakrishnan, and T. Wood. NetVM: High Performance and Flexible Networking Using Virtualization on Commodity Platforms. In *Proc. of NSDI*, Apr. 2014.

[10] IEEE Standard 802.1ag-2007: Virtual Bridged Local Area Networks, Amendment 5: Connectivity Fault Management, 2007.

[11] Intel. *Intel Data Plane Development Kit (Intel DPDK): Programmer's Guide*, October 2013.

[12] Intel LAN Access Division. PCI-SIG SR-IOV primer: An introduction to SR-IOV technology. `http://www.intel.com/content/dam/doc/application-note/pci-sig-sr-iov-primer-sr-iov-technology-paper.pdf`, January 2011.

[13] N. Katta, O. Alipourfard, J. Rexford, and D. Walker. Infinite CacheFlow in Software-Defined Networks. In *Proc. of HotSDN*, 2014.

[14] D. Katz and D. Ward. Bidirectional Forwarding Detection (BFD). RFC 5880 (Proposed Standard), June 2010.

[15] P. Kazemian, M. Chang, H. Zeng, G. Varghese, N. McKeown, and S. Whyte. Real Time Network Policy Checking Using Header Space Analysis. In *Proc. of NSDI*, 2013.

[16] P. Kazemian, G. Varghese, and N. McKeown. Header Space Analysis: Static Checking for Networks. In *Proc. of NSDI*, 2012.

[17] C. Kim, M. Caesar, A. Gerber, and J. Rexford. Revisiting Route Caching: The World Should Be Flat. In *Proc. of PAM*, 2009.

[18] K. Kogan, S. Nikolenko, O. Rottenstreich, W. Culhane, and P. Eugster. SAX-PAC (Scalable And eXpressive PAcket Classification). In *Proc. of SIGCOMM*, 2014.

[19] T. Koponen, K. Amidon, P. Balland, M. Casado, A. Chanda, B. Fulton, I. Ganichev, J. Gross, P. Ingram, E. Jackson, A. Lambeth, R. Lenglet, S.-H. Li, A. Padmanabhan, J. Pettit, B. Pfaff, R. Ramanathan, S. Shenker, A. Shieh, J. Stribling, P. Thakkar, D. Wendlandt, A. Yip, and R. Zhang. Network Virtualization in Multi-tenant Datacenters. In *Proc. of NSDI*, Seattle, WA, Apr. 2014.

[20] T. Lakshman and D. Stiliadis. High-speed Policy-based Backet Forwarding Using Efficient Multi-dimensional Range Matching. *SIGCOMM CCR*, 28(4):203–214, 1998.

[21] B.-S. Lee, R. Kanagavelu, and K. M. M. Aung. An Efficient Flow Cache Algorithm with Improved Fairness in Software-Defined Data Center Networks. In *Proc. of Cloudnet*, pages 18–24, 2013.

[22] J. Martins, M. Ahmed, C. Raiciu, V. Olteanu, M. Honda, R. Bifulco, and F. Huici. ClickOS and the Art of Network Function Virtualization. In *Proc. of NSDI*, Apr. 2014.

[23] P. E. McKenney, J. Appavoo, A. Kleen, O. Krieger, R. Russell, D. Sarma, and M. Soni. Read-copy update. In *AUUG Conference Proceedings*, page 175. AUUG, Inc., 2001.

[24] Microsoft. Hyper-V Virtual Switch Overview. `http://technet.microsoft.com/en-us/library/hh831823.aspx`, September 2013.

[25] The Netperf homepage. `http://www.netperf.org/`, January 2014.

[26] Open vSwitch – An Open Virtual Switch. `http://www.openvswitch.org`, September 2014.

[27] OpenFlow. `http://www.opennetworking.org/sdn-resources/onf-specifications/openflow`, January 2014.

[28] B. Pfaff and B. Davie. The Open vSwitch Database Management Protocol. RFC 7047 (Informational), Dec. 2013.

[29] B. Pfaff, J. Pettit, T. Koponen, K. Amidon, M. Casado, and S. Shenker. Extending Networking into the Virtualization Layer. In *Proc. of HotNets*, Oct. 2009.

[30] L. Rizzo. netmap: A novel framework for fast packet I/O. In *Proc. of USENIX Annual Technical Conference*, pages 101–112, 2012.

[31] N. Sarrar, S. Uhlig, A. Feldmann, R. Sherwood, and X. Huang. Leveraging Zipf's Law for Traffic Offloading. *SIGCOMM CCR*, 42(1), Jan. 2012.

[32] N. Shelly, E. Jackson, T. Koponen, N. McKeown, and J. Rajahalme. Flow Caching for High Entropy Packet Fields. In *Proc. of HotSDN*, 2014.

[33] S. Singh, F. Baboescu, G. Varghese, and J. Wang. Packet Classification Using Multidimensional Cutting. In *Proc. of SIGCOMM*, 2003.

[34] V. Srinivasan, S. Suri, and G. Varghese. Packet Classification Using Tuple Space Search. In *Proc. of SIGCOMM*, 1999.

[35] V. Srinivasan, G. Varghese, S. Suri, and M. Waldvogel. Fast and Scalable Layer Four Switching. In *Proc. of SIGCOMM*, 1998.

[36] V. Tanyingyong, M. Hidell, and P. Sjodin. Using Hardware Classification to Improve PC-based OpenFlow Switching. In *Proc. of High Performance Switching and Routing (HPSR)*, pages 215–221. IEEE, 2011.

[37] D. E. Taylor. Survey and Taxonomy of Packet Classification Techniques. *ACM Computing Surveys (CSUR)*, 37(3):238–275, 2005.

[38] B. Vamanan, G. Voskuilen, and T. N. Vijaykumar. EffiCuts: Optimizing Packet Classification for Memory and Throughput. In *Proc. of SIGCOMM*, Aug. 2010.

[39] VMware. vSphere Distributed Switch. `http://www.vmware.com/products/vsphere/features/distributed-switch`, September 2014.

[40] T. Y. C. Woo. A Modular Approach to Packet Classification: Algorithms and Results. In *Proc. of INFOCOM*, volume 3, pages 1213–1222 vol.3, Mar 2000.

[41] M. Yu, J. Rexford, M. J. Freedman, and J. Wang. Scalable Flow-based Networking with DIFANE. In *Proc. of SIGCOMM*, 2010.

[42] D. Zhou, B. Fan, H. Lim, M. Kaminsky, and D. G. Andersen. Scalable, high performance Ethernet forwarding with CuckooSwitch. In *Proceedings of the Ninth ACM Conference on Emerging Networking Experiments and Technologies*, CoNEXT '13, pages 97–108, New York, NY, USA, 2013. ACM.

C3: Internet-Scale Control Plane for Video Quality Optimization

Aditya Ganjam[†], *Junchen Jiang*[⋆], *Xi Liu*[†], *Vyas Sekar*[⋆],
Faisal Siddiqi[†], *Ion Stoica*[+†○], *Jibin Zhan*[†], *Hui Zhang*[⋆†]
[†]*Conviva*, [⋆]*CMU*, [+]*UC Berkeley*, [○] *Databricks*

Abstract

As Internet video goes mainstream, we see increasing user expectations for higher video quality and new global policy requirements for content providers. Inspired by the case for centralizing network-layer control, we present C3, a control system for optimizing Internet video delivery. The design of C3 addresses key challenges in ensuring scalability and tackling data plane heterogeneity. First, to ensure scalability and responsiveness, C3 introduces a novel split control plane architecture that can tolerate a small increase in model staleness for a dramatic increase in scalability. Second, C3 supports diverse client-side platforms via a minimal client-side sensing/actuation layer and offloads complex monitoring and control logic to the control plane. C3 has been operational for eight years, and today handles more than 100M sessions per day from 244 countries for 100+ content providers and has improved the video quality significantly. In doing so, C3 serves as a proof point of the viability of fine-grained centralized control for Internet-scale applications. Our experiences reinforce the case for centralizing control with the continued emergence of new use case pulls (e.g., client diversity) and technology pushes (e.g., big data platforms).

1 Introduction

Internet video is a significant and growing segment of Internet traffic today [2]. In conjunction with these growing traffic volumes, users' expectations of high quality of experience (e.g., high resolution video, low rebuffering, low startup delays) are continuously increasing [35, 3, 14]. Given the ad- and subscription-driven revenue model of the Internet video ecosystem, content providers strive to deliver high quality of experience while meeting diverse policy and cost objectives [4, 38].

In this respect, several previous efforts have shown that the observed video quality delivered by individual CDNs can vary substantially across clients (e.g., across different ISPs or content providers) and also across time (e.g., flash crowds) [39, 37]. Similarly, because the video player has only a few seconds worth of buffering and the bandwidth could fluctuate significantly, we need to make quick decisions (e.g., future bitrates) based on the current client buffer level and bandwidth so that the buffer does not drain out [27].

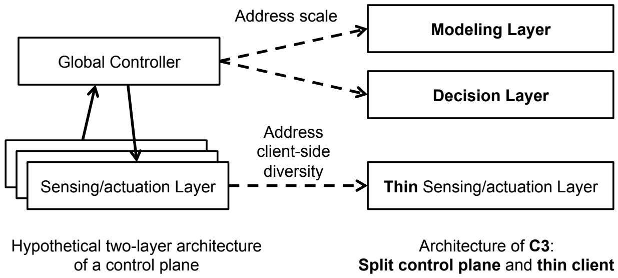

Address scale
Global Controller
Modeling Layer
Decision Layer
Address client-side diversity
Sensing/actuation Layer
Thin Sensing/actuation Layer
Hypothetical two-layer architecture of a control plane
Architecture of C3: Split control plane and thin client

Figure 1: Conceptual two-layer architecture and the ideas of split control plane and thin client

These observations have made the case for a logically centralized control plane for Internet video that uses a *global* and *real time* view of network conditions to choose the best CDN and bitrate for a given client.[1] Furthermore, content providers have complex system-wide policy and optimization objectives such as balancing costs across CDNs or servicing premium customers, which are also difficult to achieve without a global real-time view.

Conceptually, one can consider a hypothetical two-layer architecture (the left part of Figure 1) of a control plane consisting of a global controller and a client-side sensing/actuation layer. The controller uses real-time measurements of client performance observed by the sensing/actuation layer to create a *model* of expected performance, uses this model to *decide* suitable parameters (e.g., CDN and bitrate) for the clients and sends them to sensing/actuation layer to execute. Unfortunately, realizing such an Internet-scale control plane is easier said than done! For instance, at peak load we have had to handle over 3M concurrent users and we expect this to grow by up to two orders of magnitude. This *scale* makes it challenging to maintain up-to-date global views while simultaneously being responsive to client-side events.

[1] Conceptually, this can be viewed as a management layer overlaid on top of multiple CDNs and this optimization is orthogonal to the server allocation optimizations done by the individual CDNs.

To address this fundamental scaling challenge, our solution, called C3, introduces a *split control plane* architecture that logically decouples the modeling and decision functions of the controller as shown in the right part of Figure 1. The key insight underlying this unique split control plane architecture is an observation that we can tolerate a small increase in amount of staleness in the global modeling for a dramatic increase in scalability; i.e., our decisions will be close to optimal even if the global view used for decision making is out of date by a few minutes (§3).

Building on this insight, the **modeling layer** operating at a *coarse time granularity* is updated every tens of seconds or minutes to build a global model based on global view of client quality information. The **decision layer** makes real-time decisions using the global model from the modeling layer at a sub-second timescale in response to client-side events; e.g., arrivals or bandwidth drops or quality changes. Note that in contrast to traditional web serving architectures, the decision layer is not a dumb replicated caching layer but is actively making real-time decisions merging global (but stale) models with local (but up-to-date) data. An immediate consequence of the decoupling is that the decision layer interfacing with the clients is *horizontally scalable*.

Now, for any such control architecture to be effective, we need a sensing/actuation layer to (a) accurately measure video quality from video players and (b) execute the control decisions. Here, we observe a practical challenge due to *client-side diversity*. For instance, we see close to 100 distinct application combinations of framework (i.e., providing libraries to support video players) and streamer (i.e., the module responsible for downloading and rendering video). The diversity of video players coupled with practical challenges in players' long software update cycles makes it difficult to implement new measurement techniques or control algorithms. To address this client-side diversity, we make an explicit choice to make the sensing/actuation layer functionality as minimal as possible. Thus, we eschew complex control and data summarization logic in the video players in favor of a very *thin* sensing/actuation layer that exports raw quality-related events using a common data model (§4). These designs simplify adding support for new content providers, accelerate testing and integration, and also enable independent evolution of client-side platforms and C3's control logic.

Over the 8 years of operation, C3 has optimized over 100M sessions each day from over 100 name brand content providers. Our operational experience and microbenchmarks confirm that: (1) C3 controller is horizontally scalable; (2) our client-side sensing/actuation layer imposes small bandwidth overhead on the clients; and (3) C3 can dramatically improve the video quality of C3's customers within the bounds of global policies.

While C3 has evolved in response to video-specific technology trends and use-cases (§2), we believe that our lessons and design decisions are more broadly applicable to other aspects of network control (§7). First, we observe more drivers for centralized control due to greater client-side heterogeneity and more complex policy demands. Second, we see more enablers for centralizing control with the advent of big-data solutions and the ability to elastically scale service instances via cloud providers. Third, our journey reinforces the belief that separation of control and data and moving more functionality to the controller is a powerful architectural choice that enables rapid and independent evolution for different stakeholders.

2 Evolutionary Perspective

Operational systems such as C3 do not exist in vacuum—they have to constantly evolve in response to use case pulls (e.g., more complex provider policies and multi-CDN deployments) and technology trends (e.g., move from P2P to CDNs or RTMP to HTTP). In this section, we provide an evolutionary perspective of the 8-year operation of C3. This retrospective is useful because it gives us the context to understand both how the requirements (e.g., scale, diversity) have evolved and how our design decisions have adapted accordingly. We conclude with major trends that reinforce our decision to centralize the control plane.

2.1 Overview of evolution

We identify three high-level phases in the evolution of C3 (shown in Figure 2).

Phase I: The origins of C3 can be traced back to a very different operational context. The original C3 architecture was motivated by the problem of optimizing P2P live streaming. This was around 2006, when video streaming via CDNs was quite expensive with an effective cost of $\approx$ 40 cents/GB. At the time, P2P was widely perceived as an alternative low-cost solution. Unfortunately, existing overlay schemes were unreliable and unable to deliver high-quality streams equivalent to CDN-based performance. Inspired by concurrent work on the 4D architecture for network control [45], C3 was a centralized solution to manage the overlay tree in order to deliver high (CDN equivalent) quality streaming over P2P. This centralized view also enabled to implement simple per-stream global policies; e.g., limiting total bandwidth or number of viewers on a specific live channel.

During this early stage, most video streaming was based on Flash/RTMP and clients were largely homogeneous. They were largely desktop clients that needed to explicitly download/install our P2P client software, similar to other P2P systems at the time. This software would

Phases (Time)	Environments				Design overview	
	Video delivery	User scale	Platform	Policy	Key decisions	Major considerations
I (2006-2009)	P2P Live	100s-10Ks	Single	Per-stream	Centralized overlay-tree construction in the controller. Frequent update and control.	Modest size of users. Existing protocols not sufficient for high-quality streams.
II (2009-2011)	CDN, Live/VoD	1M-10Ms	Single	Global	Joint control: Controller changes the logic and bitrate/CDN list. Clients run real-time control.	Controller unable to support real-time control at scale. Flash supports dynamical loading plugin.
III (2011-now)	CDN, Live/VoD	10Ms-100Ms	Diverse	Global Complex	Minimal clients: push all decision making and quality summary to the controller.	Diverse client platforms, long software update cycle. Advent of big-data technology.

Figure 2: Overview of C3 **evolution.**

work in close coordination with the C3 controller to construct a robust overlay tree that gracefully handled user churn.

Moreover, Internet video was still in its infancy, and many premium providers were yet to step in to the market. Thus, the scale of the client demand was also relatively small. As such, C3's controller was deployed using custom server software running in dedicated datacenters. This was sufficient to provide the desired subsecond responsiveness to handle tens of thousands of clients.

Phase II: Around 2009, we saw an inflection point with several key technology and industry shifts. First, the cost of streaming using CDNs dropped significantly to ≈ 5 cents/GB. Second, many mainstream providers (e.g., iTunes, Hulu) started warming up to the potential of Internet video and started discovering monetization strategies for online video, for both live and VoD content. While Flash/RTMP still dominated as the de-facto platform for video streaming, we saw the emergence of alternatives (e.g., due to Apple refusing to support Flash). On a practical note, given that content was now being monetized, as opposed to the free video over P2P, there was some understandable reluctance from the providers to force clients to install a new client software.

These transitions had significant effects on the design of C3. First, the entry of mainstream providers meant that the workload grew several orders of magnitude from Phase I to 100s of thousands to millions users. Second, the transition to CDN-based delivery for both live and VoD meant that the C3 logic had to evolve. Specifically, the emergence of HTTP- and chunk-based video streaming protocols (e.g., [13, 8]) meant that a video client could seamlessly choose a suitable bitrate and CDN (server) at the beginning as well as in the middle of a video session with little overhead. Thus, C3 was now targeted toward the goal of better CDN and bitrate selection instead of the earlier goal of computing optimal overlay trees for live streaming.

However, our control platform was not yet mature enough to provide sub-second responsiveness at such scale. Our response in this phase was a pragmatic solution that had to sacrifice both the global view and real-time requirement to ensure the required scalability. Specifically, our solution relied on a combination of exploiting application-level resilience and clever engineering. We made a decision to coarsen the control functionality of C3. Instead of the client software, we designed a player plugin that clients would download from C3 when the video session started. This gave us coarse control wherein we could modify the player logic at the beginning of the session (e.g., choosing a CDN intelligently). For subsequent adaptation (e.g., dynamic bitrate adaptation), however, we had to rely on the local decision making and capabilities. To deal with the moderate amount of client heterogeneity, we developed custom cross-compiler techniques that allowed us to integrate our development across platforms. While this cross-compiler served us well as an interim solution, the approach soon started showing cracks as more diverse client platforms given the idiosyncrasies of different technologies.

Phase III: The current phase of C3's operation, starting in 2011, can be truly described as the coming of age of Internet video. With the ad- and subscription-driven revenue models, and the availability of rich content, many more providers and users now rely on Internet video. In fact, some industry analysts report that Internet video consumption might even exceed traditional TV.

Consequently, C3 had to evolve to once again handle 2-3 orders of magnitude increase in the client population—tens of millions clients, with 10s-100s of thousands new client arrivals per minute at peak hours. In addition, C3 now faced a more serious challenge due to client heterogeneity as we now observed very diverse client-side platforms of streaming protocols (proprietary protocols and HTTP chunking, etc), application frameworks (e.g., OSMF, Ooyala, Akamai) and devices (e.g., set-top boxes, connected TVs, tablets).

There was an independent technology shift that was synergistically aligned with these trends—the emergence of big data platforms to enable real-time processing of very large volumes of data. We embraced this technology and exploited it to enable novel solutions to handle the client-side heterogeneity. Specifically, it enabled us to make the client implementation very minimal; e.g., moving the data summarization logic originally located in the client in Phase II to the controller. This allowed us greater flexibility in adapting to new client platforms and also simplified the development cycle.

However, big-data platforms by themselves do not address the scalability challenge of providing sub-second responsiveness to client-side events for millions of clients. This required us to significantly rearchitect the control plane and motivated the split control plane architecture that we describe in the next section. Specifically, we split the controller to a geo-distributed decision layer with sub-second responsiveness exploiting the reach of cloud providers, and a consolidated modeling layer that provides minute-level freshness w.r.t. global visibility.

2.2 Major trends

The above evolution highlights two key trends that reinforce the case (in terms of both drivers and enablers) for centralizing network control:

- **Drivers:** First, we see ever increasing demands of user experience and growing complexity of the video delivery system. This naturally motivates us to move more control logic to the controller in order to use the global visibility and satisfy global policies. Second, the proliferation of diverse client platforms makes it difficult from an engineering standpoint to integrate and test the client-side logic.
- **Enablers:** With the recent advances in big data technology, we can build a backend system with unprecedented capacity to support scalable data processing in real-time with low cost. Furthermore, it is now possible to deploy the centralized control plane on-demand using cloud services with global presence. The emergence of big data platforms and cloud services can enable even greater centralized control.

In the rest of this paper, we focus on the design and implementation of the split control plane architecture and the sensing/actuation layer during this most recent phase of C3's operation.

3 C3 Split Control Plane

The goal of the C3 controller is to optimize the video quality and enforce global policies given by content providers. Note that C3 does not control the CDN servers or distribution logic. Rather it acts an additional management layer to enable content providers to achieve their quality and policy objectives on top of their existing delivery ecosystem.

There are three (arguably conflicting) goals that the C3 controller needs to meet. First, given the variability in video quality across time and space (e.g., ISP-CDN combinations) the C3 controller needs an *up-to-date global view* of network conditions to be effective in choosing a suitable CDN and bitrate for clients. Second, it needs to be *responsive* at sub-second timescales to handle new client arrivals (e.g., to minimize video startup delay) and quality-related events during video playback (e.g., drop in bandwidth or CDN congestion). Finally, and most importantly, it must be *scalable* to handle 10s-100s millions of concurrent clients.

Unfortunately, simultaneously achieving all three requirements of freshness, responsiveness and global view is hard. To see why, let us consider two strawman solutions. The first option is a single controller handling all clients. However, even state-of-the-art big data processing platforms cannot provide sub-second responsiveness with new samples arriving at the rate of 50-100 GB per minute. Even if such a system exists, there is an inherent delay to collect enough data for making decisions with high confidence; e.g., it may take minutes to infer with high confidence that a particular CDN is overloaded. A second option is to deploy replicated servers with each replica responsible for a subset of clients. Though this parallelism ensures scalability and responsiveness, the quality of the decisions will degrade as each replica will make decisions only on the partial view from its clients rather than on the global view.

Next, we present the split control plane architecture of C3 and discuss how it is crucial to simultaneously meet three key requirements—freshness, responsiveness, and global views.

3.1 Logical view

The key insight behind the split control plane is a domain-specific observation that we can slightly relax the *freshness* requirement to simultaneously achieve scalability, responsiveness, and a global view. Specifically, we observe that some global characteristics (e.g., relative rankings of CDN based on quality) are relatively stable on the order of minutes [29].

Figure 3 shows one representative result showing the *persistence* of the best CDN for clients in a given AS, which we define as the number of contiguous minute-level epochs in which this CDN has the lowest buffering ratioa cross all available CDNs. Figure 3 shows the distribution of this persistence metric across three content providers (A,B,C) that use multiple CDNs.[2] We see that the 80%ile of the persistence is 3 minutes across all three content providers.

However, such persistence does not hold for states of individual clients. For instance, when the current CDN is not available, CDN must be switched immediately to prevent the buffer from draining out (e.g.,buffer length for live videos is no more than several seconds). In this case decisions must rely on the freshest information (e.g., buffer length) to prevent quality from suffering.

The above observations on global state persistence coupled with local per-client variability make a case for a *split control plane* scheme that consists of two loosely

[2] To avoid any potential bias due to C3's control decisions, this result explicitly focuses on content providers who have not opted-in for C3's optimized control but use only the quality monitoring services.

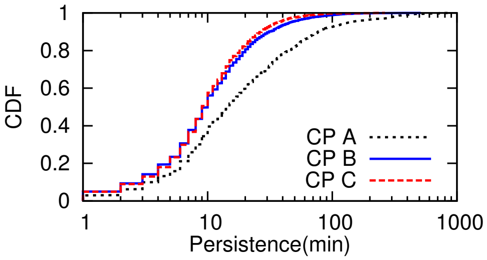


Figure 3: Distribution of the persistence of the best CDN for different content providers.

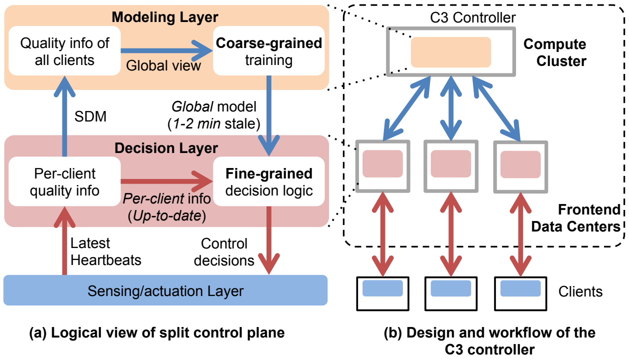
coupled stages:

- A **coarse-grained global model layer** that operates at the timescale of few tens of seconds (or minutes) and uses a global view of client quality measurements to build a data-driven prediction global model of video quality (see below).
- A **fine-grained per-client decision layer** that operates at the millisecond timescale and makes actual decisions upon a client request. This is based on the latest (but possibly stale) pre-computed global model and up-to-date per-client state.

Figure 4(a) shows how the split control plane is mapped into the two logical layers of the C3 controller. Specifically, the modeling layer implements the coarse-grained control loop (i.e., blue arrows) in a coarse timescale and trains a global model based on the measurements of each client session it receives from the decision layer. The decision layer implements the fine-grained control loop (i.e., red arrows) for each client, and makes CDN and bitrate decisions based on the global model trained by the modeling layer and latest heartbeats received from the sensing/actuation layer (see §4).

We make one important observation to distinguish the functionality of the decision layer that presents a significant departure from traditional replicated web services. Unlike traditional web services where the serving layer is a "dumb" distributed caching layer, the decision layer makes *real-time* control decisions by combining freshest quality measurements of the client under control and the global model.

This above split control plane design has two key characteristics that are critical for a *scale-out* realization of the decision layer without synchronization bottlenecks. First, note that there is a loose coupling between the fine-grained per-client and coarse-grained global control loops. Thus, we do not need the decision layer to be perfectly synchronized with the modeling layer. Second, the decision layer is operating on a per-client basis, which eliminates the need for coordination across decision layer instances. Taken together, this means that we can effectively partition the workload across clients by having a *replicated* decision layer where instances are deployed close to the clients and independently execute the logic for the subset of clients assigned to it.



(a) Logical view of split control plane

(b) Design and workflow of the C3 controller

Figure 4: Overview of the C3 controller. The SDM and Heartbeats are discussed in the next section.

3.2 End-to-end workflow

Having discussed the core ideas in the previous section, next we discuss the concrete *physical* realization of the C3 controller (shown in Figure 4(b)) and describe the end-to-end workflow.

3.2.1 Modeling layer workflow

The modeling layer is a compute cluster running a big data processing stack. The modeling layer periodically uses the information (its specific format will be introduced later in §4.2.1) collected from all clients to learn a global model that encodes actionable information useful for decision making.

Our focus in this paper is primarily on the control architecture and the design of the specific algorithms in the modeling layer is outside the scope of the paper. For completeness, we provide a high-level sketch of the algorithm. The model is similar to the nearest neighbor like prediction model suggested in prior work [39]. In particular, we leverage the insight that similar video sessions should have similar quality. Therefore, quality measurements of sessions sharing certain spatial (e.g., CDN, ISP, content provider) and temporal (e.g., time of day) features are grouped together, and intuitively, the quality of a new session can be predicted based on the quality of the most similar sessions.

To enforce global policies (e.g., traffic caps for certain CDNs), the modeling layer also includes the relevant global states as part of the global model (e.g., amount of traffic currently assigned to each CDN), so that the decision logic can take into account the global information it needs. There is a large space of potential decision logics that can take the global model, individual client state, and global policies to make optimal per-client decisions. The design of policies and algorithms to meet policy objectives is outside the scope of this paper.

The remaining question is disseminating the global model to the decision instances. Instead of a pull model like traditional web caching, the modeling layer *pushes* the global model to each frontend data center where decision instances run. The reasons for a push rather than pull approach is to ensure that each decision instance has an up-to-date model as soon as the modeling layer recomputes the global model. The overhead of the push step is quite low since the size of the global model is 100s of MBs, which can be disseminated to all data centers in several seconds without any additional optimizations and can easily fit in the memory of a modern server. (In contrast, web caches cannot know the set of requests and have to use a pull model because they cannot store the entire content catalog.)

3.2.2 Decision layer workflow

In order to minimize the response latency between clients and their corresponding decision instances, the decision instances are hosted in geographically distributed frontend datacenters as close to the clients as possible. When C3 clients arrive, they are assigned to specific decision instances via standard load balancing mechanisms, which ensure that subsequent requests (both control requests and heartbeats) of the same client are consistently mapped to the same instances. These mechanisms operate across data centers and across decision instances, and handle geographic locality, load balancing, and fault tolerance. We use industry-standard mechanisms such as DNS-based consistent mapping of clients to instances based on latency measurements. These mechanisms use standard failure detection mechanisms to detect if a specific instance has failed and reassigns clients as needed. (As shown in §4.3, the measurement collection from clients can be easily re-synced when the decision instances are reassigned.)

The clients send periodic heartbeats (described in §4.2.1) to decision instances. Based on the heartbeats of a client, a decision instance maintains a state-machine, which provides an accurate and up-to-date view of the current video quality experienced by the client. Upon receiving a control request from a client , the decision instance runs a proprietary decision algorithm to choose a suitable CDN and bitrate.

This decision logic combines both the up-to-date per-client information and the (slightly stale) global model the decision logic, and tries to optimize video quality while operating within the bounds of global polices (e.g., load per CDN or cost). In a simple example, consider two CDNs; CDN1 provides poor quality to viewers and CDN2 provides good quality in a certain city. The decision logic is able to detect that CDN1 has worse quality than CDN2 based on the quality feedback from clients using both CDNs from that particular city, and it will then instruct new clients to CDN2.

3.3 Summary of key design decisions

In summary, we make the following key decisions.

1. To balance global visibility and data freshness, we use a split control plane mechanism with a coarse timescale modeling layer loosely coupled with a fine timescale decision layer.
2. The modeling layer trains on a minute-level timescale and pushes the global model to decision layer.
3. The decision layer is horizontally scalable and can operate on a millisecond-level timescale. It combines the up-to-date per-client information, the global model, and other policies to makes optimal CDN/bitrate decisions for clients.

4 Sensing/Actuation Layer

This section presents the design of the C3 client side modules, which provide three functions. First, it reports video quality from the players to the C3 controller. Second, it receives and implements control decisions from the controller. Third, it has built-in fault tolerance when it loses connectivity to the C3 controller. There are two practical challenges in implementing these functions: (1) diversity of client-side platforms and (2) slow software update cycles of client-side platforms. We first elaborate the challenges and then describe how C3 addresses them.

4.1 Challenges

To understand the causes of the practical challenges, we need some background on the structure of the client-side platform. Each client-side platform consists of four key components: client operating system, application framework, streamer, and player application. The application framework runs on top of the operating system and provides the libraries to support the development of video player applications. Many application frameworks can run atop the same operating system and device hardware. The streamer is responsible for downloading and rendering the video. Finally, the player application is the software developed by a content publisher based on specific application framework, to implement the user interface, access to content library, and player navigation.

Diversity: We observe client diversity along several dimensions; e.g., programming language (e.g., C, Javascript, Lua), system support for code execution (e.g., support for multi-threading), application framework, and streamer. The diversity of application frameworks and streamers is especially critical as it defines the interfaces used to monitor and control the video quality, and specifies the programming environment. Table 1 shows three examples of operating systems and devices and a subset of application frameworks. In total, we encounter 95 distinct application framework and streamer combinations. Each such combination requires special attention to mon-

OS	Devices	Application Frameworks
Android OS	Android phones/tablets	MediaPlayer, Irdeto, NexStreaming, Video View, VisualOn, PrimeTime, Akamai
PlayStation OS	PS3, PS4, PS Vita	Trilithium, LibJScript, WebMAF, Touchfactor
Mac OS & Windows with Flash	PCs	OSMF, Kaltura, Ooyala, PrimeTime, Brightcove, FlowPlayer, ThePlatform, JWPlayer
iOS	iPhone, iPad, iPod Touch	AVFoundation, Ooyala, Brightcove, PrimeTime, MediaPlayer, Irdeto

Table 1: Examples of OS and devices with the corresponding application frameworks.

itoring and control interfaces. Such diversity further reinforces the need to minimize the client-side functionality.

Long software update cycles: The second major challenge is long software update cycles for client-side platforms. There are a host of contributing factors here; e.g., device firmware update cycle (3-12 months), publisher app updates (1-6 months) and app store ratifications (1-4 weeks), and user delays in applying updates (weeks to months). Unlike in Phase II where Flash/RTMP platforms support a player module to be downloaded dynamically, the integration code in most Phase-III platforms is embedded in the player binary and cannot be changed arbitrarily. This long update cycle fundamentally constrains the pace of evolution of the C3 platform with respect to any functions that depend on sensing/actuation layer. Although the decision algorithms are not constrained by the update cycle as it is already on the controller, it does impact the information available to the control logic. For instance, if some quality metric is currently not collected or cannot be extrapolated from the collected information, the control logic will not be able to use it until the next release, which as we have see can take months or even a year (e.g., set-top boxes).

4.2 Thin Client-Side Design

Next, we discuss how we address the challenge of client diversity and long update cycles by making the sensing/actuation layer functionality as *minimal* as possible. We do so via two key design decisions. First, we introduce a general and abstract representation of video player actions. This allows us to handle client diversity. Second, we make an explicit decision to export raw events rather than summary statistics and push this computation to the backend. This delayed binding enables us to tackle the uncertain software upgrade cycles in the wild.

4.2.1 Abstracting player state and control

To minimize the amount of engineering effort required to support the client-side heterogeneity, we identify a logical narrow waist that we call the *ConvivaStreamingProxy (CSP)* (Figure 5). CSP abstracts away the idiosyncrasies of different players, and implements high-level monitoring APIs for collecting player performance informa-

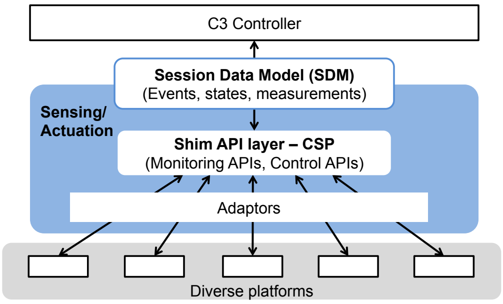


Figure 5: Overview of the sensing/actuation layer.

tion, and control APIs for switching bitrate and CDN. For each unique pair of streamer and application framework that we want to integrate, we implement an *adaptor* using the CSP API. The adaptor translates between the framework/streamer-specific APIs and the CSP APIs.

While the adaptor is specific to each framework and streamer, the common logic above the CSP is reusable across different platforms. To reduce the engineering efforts and support diverse programming languages used by the application, we developed a custom language translator that can take the source code from one language (in this case C#) and generate the equivalent source code for other languages. The design of this translator is outside the scope of this paper.

The CSP uses a unified monitoring interface, called *Session Data Model (SDM)* between clients and the C3 controller (Figure 5). SDM is a conceptual model for Internet video sessions and is agnostic to device, OS, application framework, or streamer features. Consequently new platforms can be integrated with little change on the controller. The SDM defines events, states and measurements as following:

- Events encode one-time actions and may change a state variable. Examples are bitrate switch start/end, application error.
- States encode persistent state variables, such as player state (buffering, playing, etc), bitrate and CDN.
- Measurements are continuous variables that show the health of the player, such as frame rate, available bandwidth, and buffer length.

CSP also provides the control APIs between clients and the controller. The clients send *poll* requests to get control decisions of bitrate and CDN at well-defined intervals (e.g., either at periodic intervals or at video chunk boundaries).

4.2.2 Exposing raw data

While the SDM abstraction minimizes the effort in integrating new platforms, it does not address the other practical challenges arising from long software update cycles. Specifically, this means that some logic (e.g., quality metric computation) becomes inflexibly hardcoded in client side. In order to reduce the need to make changes to clients, we build on top of the SDM abstraction and instrument the client to *report the raw events and player states*. For example, we could calculate the average bitrate of a session on the client and send this to the controller. In contrast, our approach reports all bitrate switch events to the backend and allow it calculate the average bitrate (or any other bitrate-related metric). This *delayed binding* in postponing summarization of quality metrics to the controller further embodies the high-level decision to make the client-side as minimal as possible.

The frequency at which the clients report the controller naturally induces a tradeoff between overhead and information freshness. On one hand, the clients should report quality frequently so that the controller can detect client-side events (e.g., session exit, buffer draining out) in time and make decisions accordingly. On the other hand, updating too frequently may overload the controller and/or consume too much client-side resources. To address this problem, we take the following practical approach. The sensing/actuation layer periodically batches the collected information into *heartbeats* before sending them to the controller. In practice, we choose a sweet spot between 5 seconds and 20 seconds; intervals ≤ 5 seconds introduce undesirable interactions especially on mobile devices (e.g., draining battery by increasing CPU and radio use) and intervals ≥ 20 seconds significantly reduce freshness of data used in decision making. Fortunately, most playback buffers are on the order of 30 seconds, so the controller is always able to react before the buffer drains out. Additionally, the controller can dynamically tune the reporting frequency; e.g., decreasing the frequency during flash crowds to reduce the overhead and increasing the frequency for a client with a low buffer.

4.3 Fault tolerance

The main failure mode is when the client can no longer contact one of the C3 servers implementing the decision layer functions; e.g., the server failed or the network link is unreliable. There are two potential concerns we need to address: (1) loss in quality (because the client cannot receive control decisions) and (2) information loss (because the client cannot send quality measurements). Fortunately, we can leverage application-level resilience in conjunction with the SDM to address both issues.

First, we ensure that client-perceived quality will degrade gracefully when the C3 controller is unreachable. Because there is no tight coupling between the client and the decision layer, we can handle decision layer failures by simply resending requests and reports, and allow the load balancer to reassign the client to a new server. If the client is unable to contact the controller, it will fall back to the native bitrate adaptation algorithms [30, 9], which most platforms support today. The native algorithms are able to select bitrate with local logic (e.g., using throughput or buffer occupancy), so the player can still provide descent quality of experience.

Second, to mitigate the impact of information loss, we use built-in resilience provided by the SDM semantics because it explicitly includes current player states. To see why, consider an alternative solution that only reports the events without reporting current player states. The problem is that even a single lost event can mislead the controller; e.g., if we miss an event where the player transitioned from playing to buffering state, the controller will incorrectly assume that it is currently in playing state. However, we can mitigate the impact of lost heartbeats by including a snapshot of current states in each heartbeat. When a heartbeat is lost or a C3 server is down (when all history events are lost), even though the controller cannot recover lost events, the new C3 server can infer the current state using the next heartbeat.

4.4 Summary of key design decisions

In summary, the C3 client-side component has the following key aspects:

1. A common data/control abstraction via the SDM interface to tackle client-side diversity.
2. Exposing raw data to address slow client release cycle.
3. Providing a configurable reporting frequency to reduce the overhead.
4. Enabling graceful degradation by falling back to the native adaptation algorithm to handle transient failures and using stateful SDM features to re-establish context when raw events are lost.

5 Evaluation

In this section, we evaluate the performance of C3 and the benefits it offers. We divide our evaluation into the following parts;

- We evaluate the C3 controller in terms of (a) scalability and responsiveness of the decision layer (§5.1) and (b) the ability of the modeling layer to handle various workloads within the deadlines (§5.2).
- We show the sensing/actuation layer is lightweight in terms of bandwidth consumed and can gracefully degrade user experience under failures (§5.3).
- We analyze the quality improvement that C3 offers in the wild and discuss anecdotal experiences in handling high impact events (e.g., FIFA World Cup) (§5.4).

5.1 Decision layer

Scalability: By design, the decision layer is horizontally scalable (§3), with no synchronization needed between its instances to handle client requests.

Here, we focus on evaluating the *requests per second* (RPS) that a single decision instance can process within a given response time threshold. The instance under test is

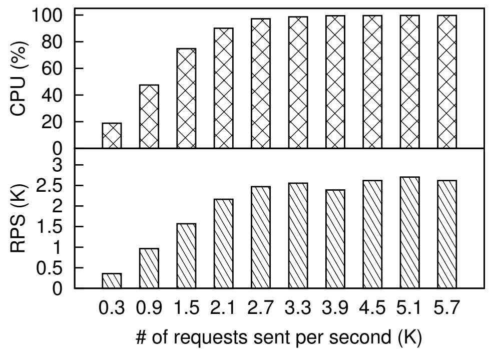


Figure 6: Scalability of single decision instance.

based on an Intel Xeon L5520 2.27Gz with 4 GB RAM, and only one core is used. Note that the requests include heartbeats and control requests, both of which are processed via the same procedure.

Figure 6 shows the RPS and CPU utilization when receiving different number of requests. It shows a linear growth in RPS and CPU utilization before the instance is saturated at the point of 2500 requests.

Load balancing: Next, we use real world measurements to show that the requests assigned by the load balancer to each decision instance is evenly distributed, even under high load. Figure 7 shows the distribution of RPS and CPU utilization of decision instances in different decision data centers, under a high load during the one of the most popular games in World Cup 2014. The means and standard deviations are based on all instances in each data center. It shows that the load balancer can assign the requests almost evenly across all decision instances within each data center with little variance and no request being dropped. The differences across the data centers is because the wide-area load balancing used geographical proximity and the workload was unevenly distributed around the world. Finally, the low CPU utilization suggests this load is well below the instance capacity.

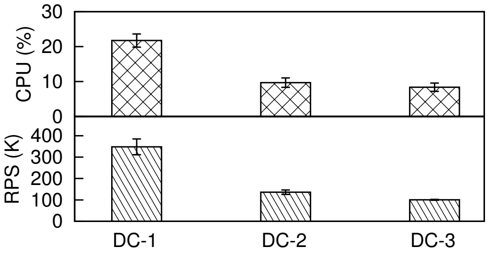


Figure 7: Number of requests and CPU utilization (mean and standard error) of decision instances under a high load.

End-to-end response time: The key metric of interest for the decision layer is the end-to-end response time – the time between client sends a request and it receives the response (typically a control decision). It includes internal processing latency as well as network latency. We measured the response time of requests in the real production system. Figure 8 presents the response time of requests observed from 10 countries representing different continents. It does show variance among different countries, but overall, we observe the 50%ile (or 80%ile) is always below 400ms (or 800ms)

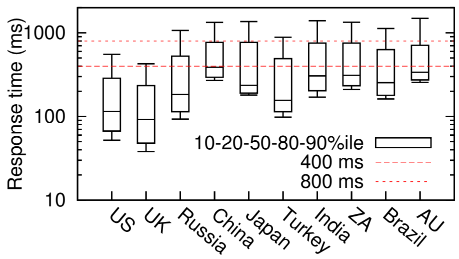


Figure 8: Response time is consistently low across countries from different continent.

5.2 Modeling layer

The modeling layer has to process various processing jobs with different characteristics. Figure 9 shows the processing latency of three most typical types of jobs and compares them with their deadlines, i.e., maximum expected processing time. The global model ("GlobalTraining") needs to be refreshed every minute and needs to run complex machine learning algorithms over the recent (few minutes) of global measurements. Customer interactive queries ("InteractiveQuery") run on-demand, and have to be completed in sub-second response time to prevent the customers from waiting too long. Finally, live update of quality metrics ("LiveUpdate") is the metric computation and aggregation process and has to be sub-second. As shown in Figure 9, the processing latency can easily satisfy the deadlines of different jobs.

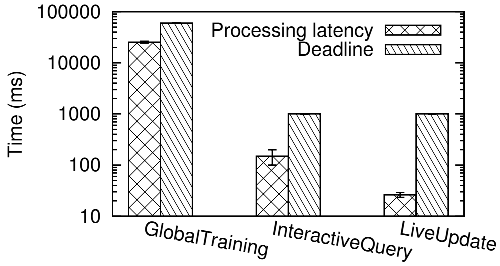
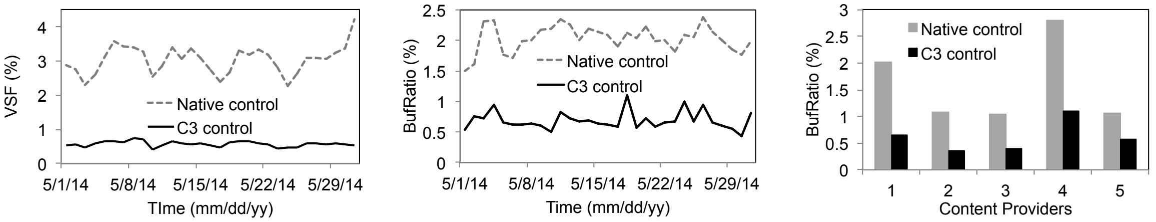


Figure 9: Processing latency (mean and standard deviation) of different types of jobs in modeling layer compared to their deadlines.

5.3 Sensing/actuation layer

Next, we show that sensing/actuation layer has relatively light overhead compared to player execution and it provides local failover under decision layer failures.

Overhead of sensing/actuation layer: We evaluate the sensing/actuation layer overhead in terms of the additional bandwidth used by comparing a C3-enabled



(a) Video start failure rate (VSF).
(b) Buffering ratio (BufRatio).
(c) Buffering ratio of 5 content providers.

Figure 11: Quality improvement of using C3**.**

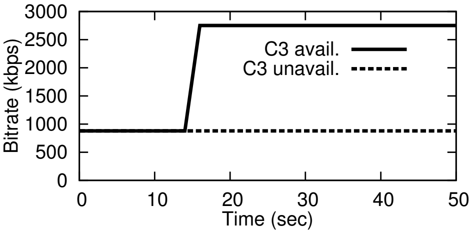


Figure 10: Local application-level resilience when C3 **controller is unavailable.**

player and a base video player. The base player we use is a fully functional player with OSMF as its streamer, and we let both the base player and C3-enabled version play a video encoded in 1.5Mbps bitrate on a laptop running Windows OS. We find that the additional bandwidth used by C3 is are typically very low and $\leq 1\%$ of the bandwidth used to download the video (not shown).

Local failover under decision layer failures: Next, we stress test the client under the scenario when it loses communication with the C3 controller. By design (§4.3), the player should fall back to the player's native adaptive bitrate logic and play smoothly; i.e., as long as the available bandwidth can sustain the lowest bitrate, the player should play smoothly, though with a low quality. We set up a real player to play a video encoded in multiple bitrates (from 880 to 2750 kbps), and the content is available from two CDNs. Figure 10 shows the time-series of bitrate downloaded by the player over two runs. In the first run, the C3 controller is not available and we throttle the bandwidth to the default CDN, so that the player backoffs to the native control logic and sticks to the low bitrate but not crash. In the second run, the C3 controller is available, and we again throttle the bandwidth to the same CDN. This time, because the C3 controller identifies the performance difference of two CDNs, it instructs the player to start with the unthrottled CDN, and thus it is able to switch to higher bitrate after a few chunks.[3]

5.4 C3 real-world deployment

Finally, we present the quality improvement C3 offers in the wild and our experiences in managing some high-impact events.

5.4.1 Quality improvement

We begin with the real-world quality improvements brought by centralizing decision logic detailed in §3.2.

Benefits of using C3**:** To estimate the quality improvement, we did a randomized trial where each session was randomly assigned to use either C3 or the native control logic. Figure 11 shows the improvement during May 2014, in terms of two quality metrics and across five content providers by comparing the median quality of sessions using C3 vs. native clients.

First, Figure 11(a) shows that C3 can significantly reduce the video start failure rate (percent of video sessions that failed to start) of a content provider. The reason is that using the global view allows us to predict CDN performance and choose an initial CDN that is not overloaded, unlike the native logic, which is typically statically configured (or chosen at random). Second, Figure 11(b) compares the buffering ratio of a content provider, and it shows a consistent reduction by 50% in buffering ratio by C3. The reason is that we can adapt the midstream selection of bitrate and CDN by leveraging the quality information of other sessions to achieve higher bitrate and a lower buffering ratio. Finally, Figure 11(c) shows that the quality improvement is consistent across five different content providers that use C3.

Impact of data staleness: Next, we present a trace-driven what-if analysis to quantify how much the decrease in freshness can impact the optimality of such quality improvement. Recall that allowing the global view to be stale on the order of few minutes was a key enabler for the scale-out design in §3. To avoid any biases introduced by our own control logic, we use the trace of sessions of Feb 10 from a content provider whose decisions are not controlled by C3 (as in §3.1). We simulate the effect of selecting the best observed CDN t minutes ago for each AS, and vary the degree of staleness by modifying this observation window t. Ideally, the decisions are made using the most recent view, i.e., $t = 1$. We evaluate multiple levels of staleness with $t = \{2, 10, 20, 40, 80, 160\}$ minutes. For each t, we compare *FreshestDecision* (i.e., times of picking the actual best CDN based on the freshest data) and *StaleDecision* (i.e., times of picking the actual best CDN based on the

[3] The bitrates of the initial chunks are intentionally chosen to be low to minimize join time and avoid early buffering.

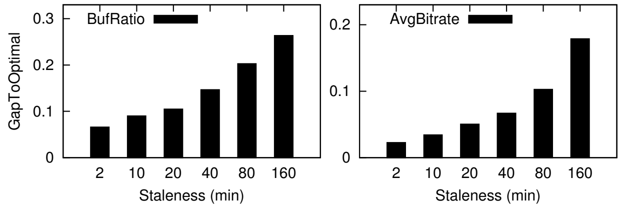


Figure 12: Impact of using stale data as input.

t-minute stale data), and compute the $GapToOptimal = 1 - \frac{StaleDecision}{FreshestDecision}$. The impact of staleness is smaller when $GapToOptimal$ is closer to zero.

Figure 12 shows the $GapToOptimal$ of buffering ratio and average bitrate under different staleness t. First of all, it shows that using 2-minute stale data has very little impact, increasing buffering ratio by less than 7% and average bitrate by less than 2.5%. This suggests that minute-level stale data is still useful to make near-optimal decisions. Second, in both metrics, $GapToOptimal$ increases slowly when t increases from 2 to 20 minutes, and begins to increase much faster from $t = 40$. This suggests that near-optimal decisions can be made with slightly stale global model.

5.4.2 Case studies with high-impact events

In this section, we focuses on case studies on high-impact events (e.g., popular sports events) to showcase that the design of C3 is flexible and scales out horizontally.

Scale-out capability: The first case study highlights the flexibility of the scale-out design that enabled us to invoke public resources on demand. During World Cup 2014, we provisioned additional decision instances capability, including instances from a major public cloud service provider, in order to handle the scale of clients, which was expected to be 6M concurrent viewers. As a result, we were able to successfully handle the peak of 3.2M concurrent viewers during the US-Germany game. The reason for this scale-out capability is that the client-facing decision layer is decoupled from the modeling layer, which is the only centralized function (§4.2).

Dynamic reconfiguration: The second case is an example of flexibility that enables C3 to drop certain functionality under unexpected flash crowd. During a very popular soccer game of one European soccer league we saw about 2M concurrent viewers. Specifically, we saw a large flash crowd when the content provider switched all of its viewers to another channel, causing a high join rate which exceeded the capacity of our hardware load balancer for doing SSL offload. Since we could not add capacity to the hardware load balancer, we needed to devise quick workarounds to reduce this load. Our solution was to reduce the heartbeat frequency and disable HTTPS for that particular content provider. This effectively reduced the overhead of per-session operation (e.g., SSL handshakes) and ensured the availability of C3 controller. This type of fast reaction to unexpected flash crowds was possible because we had a thin client and had moved most of the functions to the C3 controller—it would have been virtually impossible to reconfigure client behaviors with hardcoded client-side player logics.

6 Related Work

C3 is related to a rich body of work in network control, application-layer systems, and big data platforms. While C3 borrows and extends ideas from these relevant communities, the core contribution of our work is in synthesizing these ideas to demonstrate the feasibility of an Internet-scale control plane architecture for Internet video. We briefly describe key similarities and differences between C3 and related work.

Network control plane: As discussed earlier in §2, the origins of C3 were inspired by the precursors to SDN (e.g., [45, 18, 43, 33, 19]) and in many ways C3's evolution has paralleled the corresponding rise of SDN. As such, there are natural analogies between C3 and SDN, in terms of the sets of challenges that both have to address: interface between control and data plane (e.g., [41, 17], distributed and global state management (e.g., [33, 15, 36]), consistency semantics (e.g., [40, 44]), centralized optimization algorithms (e.g., [26, 28]). The key differences are that C3 focuses on a specific video application ecosystem, which entails different domain-specific challenges (e.g., larger scale of clients and more data plane heterogeneity) and domain-specific opportunities (e.g., weaker consistency).

Video quality optimization: Previous work confirms that video quality impacts user engagement [23, 14]. It also identifies that many of the quality issues today are a result of sub-optimal client-side control logic (e.g., [27, 30]), and spatial and temporal diversity of performance across different CDNs and content providers [39, 38, 29], which suggests a centralized controller or a federated architecture [16] that provides global state and enables better informed decision making. However, these prior studies made a case for centralized control but fell short of actually demonstrating the viability of that control plane or the real benefits in the wild. In contrast, C3 is a concrete production design and implementation that achieves the benefits identified by these efforts. In doing so, it addresses many challenges (e.g., scalability, fault tolerance) that these prior works did not try to address.

Application resilience: The idea of exploiting domain- and workload-specific insights for improving system scalability and resilience is far from new and has been repeatedly identified; e.g., both in Internet services [24, 31, 5] and distributed file systems [25, 20]. For instance GFS exploits a unique workload pattern [25] while Spanner exploits tolerance for weaker consistency [20]. Our specific contribution is in reinforcing this insight in the context of an Internet-scale control plane architecture.

Real-time data processing systems: In some sense, C3 can also be viewed as an instance of a scale out analytics and control system in the spirit of other big-data solutions (e.g., [21, 6, 22, 12, 11, 34, 10, 1]). Indeed, the C3 implementation builds on (and has actively contributed to) a subset of these existing technologies. While the C3 implementation relies on tools such as Spark [11] and Kafka [34], the core ideas are quite general and can be ported to other platforms. At a conceptual level, C3 also shares some similarity with the broad purview of the recent Lambda Architecture [7], with a combination of batch, serving, and speed layers. More recent work on Velox [21] also recognized the separation of global modeling and per-client prediction as a powerful system design choice for other data analytics applications. While C3 follows a similar high-level multi-layer architecture, the key is the specific division of functions between layer for video quality optimization. For instance, unlike the front-end layer in most scale-out systems, C3's front-end decision layer is an active layer that runs decision-making functions as well. Moreover, C3 justifies the separation of modeling and decision making by the domain-specific observation that slightly stale global models can still achieve near-optimal decisions.

7 Lessons

We conclude with key lessons we have learned from building and operating C3. Even though C3's design was driven by video-centric challenges and opportunities, these lessons have broader implications to concurrent and future efforts for centralized network control.

Feasibility of Internet-scale control: The one obvious lesson from our journey is that it is indeed possible to implement an Internet-scale control platform that achieves policy goals with global visibility. While there are parallel SDN success stories demonstrating the viability of centralized control, these have been in different (and arguably more scoped) domains; e.g., low-latency datacenters [32, 42], wide-area networks with a few PoPs [28] or coarse-grained inter-domain route control [18]. With C3, we provide another proof point in what we believe to be a much more global deployment, with larger scale and more heterogeneous clients, more stringent policy, and greater expectations in quality of experience.

Decisions that worked: Among the many decision choices that have made C3 a proof point of Internet-scale control, we highlight three that were particularly useful:

- **Exploiting application-level resilience:** A key enabler for our scale-out control architecture is that we were able to appreciate and exploit domain-specific properties that allows us to weaken some requirements (e.g., consistency and model freshness). While this idea may not be new and has been re-observed in many contexts, we believe that it is especially useful for network control applications. For instance, there has been considerable research effort developing consistent update schemes for SDN [40]. Rather than building a general-purpose solution, it might be possible to leverage application-specific resilience and engineer simpler schemes with weaker consistency properties.
- **Minimal client functionality:** We cannot stress enough the advantages that we have derived from minimizing the client functionality. This has (i) dramatically simplified our product development, integration cycles to support the increasingly heterogeneous client-side platforms, and (ii) made the C3 controller flexible to enforce global policy and evolve. The minimal client design is also made possible by the increasing compute capabilities of the backend.
- **Exposing lower-level APIs:** While minimal client functionality is useful, it has also been proved surprisingly useful to expose as many lower-level APIs from clients as possible, since they maximize control logic extensibility with a (relatively) slowly evolving data plane. We see immediate implications of this in SDN. While SDN started off with a minimal API (e.g., early OpenFlow versions), it soon devolved into the same complexity pitfalls that it sought to avoid (e.g., OpenFlow 1.3 spec is 106 pages long [17]). We believe it might be worthwhile for the SDN community to revisit minimality in light of the benefits we have derived and we already see early efforts to this end [17].

New research opportunities: C3's design and deployment opens up new possibilities for video quality optimization. The current C3 design is only a step in our journey and we acknowledge that there are several directions for future work that we do not cover in this paper. For example, one interesting question is analyzing the interaction of C3 with CDN control loops. In terms of quality improvement, we need a better understanding of clients that show little improvement and techniques to leverage network and CDN information for better quality diagnosis. Similarly, there are interesting modeling and algorithmic questions in the design of the prediction modeling and decision algorithms that have significant room for improvement.

Acknowledgments

This paper would not be possible without the contribution of the Conviva staff. Junchen Jiang was supported in part by NSF award CNS-1345305. The authors thank Sachin Katti for shepherding the paper and the NSDI reviewers for their feedback.

References

[1] Apache flume. incubator.apache.org/flume.

[2] Cisco forecast. http://blogs.cisco.com/sp/comments/cisco_visual_networking_index_forecast_annual_update/.

[3] Cisco study. http://www.cisco.com/web/about/ac79/docs/sp/Online-Video-Consumption_Consumers.pdf.

[4] Driving Engagement for Online Video. http://events.digitallyspeaking.com/akamai/mddec10/post.html?hash=ZDlBSGhsMXBidnJ3RXNWSW5mSE1HZz09.

[5] Facebook scribe. github.com/facebook/scribe.

[6] Hadoop. http://hadoop.apache.org/.

[7] Lambda archictecture. lambda-architecture.net.

[8] Microsoft Smooth Streaming. http://www.microsoft.com/silverlight/smoothstreaming.

[9] Netflix. www.netflix.com/.

[10] S4 distributed stream computing platform. incubator.apache.org/s4.

[11] Spark. http://spark.incubator.apache.org/.

[12] Storm. storm-project.net.

[13] I. Sodagar. The MPEG-DASH Standard for Multimedia Streaming Over the Internet. *IEEE Multimedia*, 2011.

[14] A. Balachandran, V. Sekar, A. Akella, S. Seshan, I. Stoica, and H. Zhang. Developing a predictive model of quality of experience for internet video. In *ACM SIGCOMM '13*.

[15] P. Berde, M. Gerola, J. Hart, Y. Higuchi, M. Kobayashi, T. Koide, B. Lantz, B. OConnor, P. Radoslavov, W. Snow, et al. Onos: towards an open, distributed sdn os. In *ACM HotSDN 2014*.

[16] A. Biliris, C. Cranor, F. Douglis, M. Rabinovich, S. Sibal, O. Spatscheck, and W. Sturm. Cdn brokering. *Computer Communications*, 2002.

[17] P. Bosshart, D. Daly, G. Gibb, M. Izzard, N. McKeown, J. Rexford, C. Schlesinger, D. Talayco, A. Vahdat, G. Varghese, et al. P4: Programming protocol-independent packet processors. *ACM SIGCOMM CCR*, 2014.

[18] M. Caesar, D. Caldwell, N. Feamster, J. Rexford, A. Shaikh, and J. van der Merwe. Design and implementation of a routing control platform. In *NSDI 2005*.

[19] M. Casado, M. J. Freedman, J. Pettit, J. Luo, N. McKeown, and S. Shenker. Ethane: Taking control of the enterprise. *ACM SIGCOMM CCR 2007*.

[20] J. C. Corbett, J. Dean, M. Epstein, A. Fikes, C. Frost, J. Furman, S. Ghemawat, A. Gubarev, C. Heiser, P. Hochschild, et al. Spanner: Googles globally distributed database. *ACM Transactions on Computer Systems (TOCS) 2013*.

[21] D. Crankshaw, P. Bailis, J. E. Gonzalez, H. Li, Z. Zhang, M. J. Franklin, A. Ghodsi, and M. I. Jordan. The missing piece in complex analytics: Low latency, scalable model management and serving with velox. In *Conference on Innovative Data Systems Research (CIDR)*, 2015.

[22] J. Dean and S. Ghemawat. Mapreduce: Simplified data processing on large clusters. *OSDI 2004*.

[23] F. Dobrian, V. Sekar, A. Awan, I. Stoica, D. A. Joseph, A. Ganjam, J. Zhan, and H. Zhang. Understanding the impact of video quality on user engagement. In *Proc. SIGCOMM*, 2011.

[24] M. J. Freedman. Experiences with coralcdn: A five-year operational view. In *NSDI 2010*.

[25] S. Ghemawat, H. Gobioff, and S.-T. Leung. The google file system. In *ACM SIGOPS Operating Systems Review 2003*.

[26] C.-Y. Hong, S. Kandula, R. Mahajan, M. Zhang, V. Gill, M. Nanduri, and R. Wattenhofer. Achieving high utilization with software-driven wan. In *ACM SIGCOMM 2013*.

[27] T.-Y. Huang, R. Johari, N. McKeown, M. Trunnell, and M. Watson. A buffer-based approach to rate adaptation: evidence from a large video streaming service. In *ACM SIGCOMM 2014*.

[28] S. Jain, A. Kumar, S. Mandal, J. Ong, L. Poutievski, A. Singh, S. Venkata, J. Wanderer, J. Zhou, M. Zhu, et al. B4: Experience with a globally-deployed software defined wan. In *ACM SIGCOMM 2013*.

[29] J. Jiang, V. Sekar, I. Stoica, and H. Zhang. Shedding light on the structure of internet video quality problems in the wild. In *ACM CoNEXT 2013*.

[30] J. Jiang, V. Sekar, and H. Zhang. Improving Fairness, Efficiency, and Stability in HTTP-Based Adaptive Streaming with Festive . In *ACM CoNEXT 2012*.

[31] L. Kontothanassis, R. Sitaraman, J. Wein, D. Hong, R. Kleinberg, B. Mancuso, D. Shaw, and D. Stodolsky. A transport layer for live streaming in a content delivery network. *Proceedings of the IEEE*, 92(9):1408–1419, 2004.

[32] T. Koponen, K. Amidon, P. Balland, M. Casado, A. Chanda, B. Fulton, I. Ganichev, J. Gross, N. Gude, P. Ingram, et al. Network virtualization in multi-tenant datacenters. In *NSDI 2014*.

[33] T. Koponen, M. Casado, N. Gude, J. Stribling, L. Poutievski, M. Zhu, R. Ramanathan, Y. Iwata, H. Inoue, T. Hama, et al. Onix: A distributed control platform for large-scale production networks. In *OSDI 2010*.

[34] J. Kreps, N. Narkhede, and J. Rao. Kafka: A distributed messaging system for log processing. In *Proceedings of the NetDB*, 2011.

[35] S. S. Krishnan and R. K. Sitaraman. Video stream quality impacts viewer behavior: inferring causality using quasi-experimental designs. In *ACM IMC*, 2012.

[36] D. Levin, A. Wundsam, B. Heller, N. Handigol, and A. Feldmann. Logically centralized?: state distribution trade-offs in software defined networks. In *ACM HotSDN 2012*.

[37] H. Liu, Y. Wang, Y. R. Yang, A. Tian, and H. Wang. Optimizing Cost and Performance for Content Multihoming. In *Proc. SIGCOMM*, 2012.

[38] H. H. Liu, Y. Wang, Y. R. Yang, H. Wang, and C. Tian. Optimizing cost and performance for content multihoming. *ACM SIGCOMM CCR*, pages 371–382, 2012.

[39] X. Liu, F. Dobrian, H. Milner, J. Jiang, V. Sekar, I. Stoica, and H. Zhang. A Case for a Coordinated Internet Video Control Plane. In *SIGCOMM*, 2012.

[40] R. Mahajan and R. Wattenhofer. On consistent updates in software defined networks. In *ACM HotNets 2013*.

[41] N. McKeown, T. Anderson, H. Balakrishnan, G. Parulkar, L. Peterson, J. Rexford, S. Shenker, and J. Turner. Openflow: enabling innovation in campus networks. *ACM SIGCOMM CCR*, 38(2):69–74, 2008.

[42] J. Perry, A. Ousterhout, H. Balakrishnan, D. Shah, and H. Fugal. Fastpass: a centralized zero-queue datacenter network. In *ACM SIGCOMM 2014*.

[43] B. Raghavan, M. Casado, T. Koponen, S. Ratnasamy, A. Ghodsi, and S. Shenker. Software-defined internet architecture: decoupling architecture from infrastructure. In *ACM HotNets '12*.

[44] P. Sun, R. Mahajan, J. Rexford, L. Yuan, M. Zhang, and A. Arefin. A network-state management service. In *ACM SIGCOMM 2014*.

[45] H. Yan, D. A. Maltz, T. E. Ng, H. Gogineni, H. Zhang, and Z. Cai. Tesseract: A 4d network control plane. In *NSDI*, volume 7, pages 27–27, 2007.

Attaining the Promise and Avoiding the Pitfalls of TCP in the Datacenter

Glenn Judd
Morgan Stanley

Abstract

Over the last several years, datacenter computing has become a pervasive part of the computing landscape. In spite of the success of the datacenter computing paradigm, there are significant challenges remaining to be solved—particularly in the area of networking. The success of TCP/IP in the Internet makes TCP/IP a natural candidate for datacenter network communication. A growing body of research and operational experience, however, has found that TCP often performs poorly in datacenter settings. TCP's poor performance has led some groups to abandon TCP entirely in the datacenter. This is not desirable, however, as it requires reconstruction of a new transport protocol as well as rewriting applications to use the new protocol. Over the last few years, promising research has focused on adapting TCP to operate in the datacenter environment.

We have been running large datacenter computations for several years, and have experienced the promises and the pitfalls that datacenter computation presents. In this paper, we discuss our experiences with network communication performance within our datacenter, and discuss how we have leveraged and extended recent research to significantly improve network performance within our datacenter.

1 Introduction

In recent years, datacenter computing has become a pervasive part of the computing landscape. The most visible examples of datacenter computing are the warehouse-scale computers [4] used to run search engines, social networks, and other publicly visible "cloud" applications. Less visible, but no less critical, are datacenter computing platforms used internally by numerous organizations.

In spite of the success of the datacenter computing paradigm, there are significant challenges remaining to be solved—particularly in the area of networking. The pervasiveness of TCP/IP in the Internet makes TCP/IP a natural candidate for datacenter network communication. TCP/IP, however, was not designed for the datacenter environment, and many TCP design assumptions—e.g. a high degree of flow multiplexing, multi-millisecond RTT—do not hold in a datacenter. A growing body of research and operational experience, has found that TCP can perform poorly in datacenter settings.

TCP's poor performance has led some groups to abandon TCP entirely [15]. This is not desirable, however, as it requires reconstruction of a new transport protocol as well as rewriting applications to use the new protocol. Recent research has focused on adapting TCP to operate in the datacenter environment. DCTCP stands out as a particularly promising approach as it utilizes technology available today to dramatically improve datacenter TCP performance.

In this paper, we discuss our experiences with network communication performance within our datacenter and discuss how we have leveraged and extended recent research to significantly improve network performance within our datacenter, without requiring changes to our applications.

The experimental results that we present are often in the form of controlled tests that isolate behavior that we encountered either in actual production TCP and DCTCP usage, or in our efforts to introduce DCTCP into production.

In addition, this paper makes the following specific contributions.

- To the best of our knowledge, this paper presents the first published discussion of DCTCP production deployment.
- We identify shortcomings that make DCTCP as presented and implemented in [1] unusable in our environment, and we present solutions to those shortcomings that we have verified through implementation.
- We demonstrate that commonly used receive buffer tuning algorithms perform poorly in current datacenters.
- We empirically compare DCTCP performance to TCP convergence, and we show that—surprisingly—DCTCP convergence can be superior to TCP convergence. We show that this is due to DCTCP's superior coexistence with common receive buffer tuning algorithms. With correct buffer tuning, TCP convergence, stability, and short-term fairness all exceed that of DCTCP.
- We also discuss results from dramatically reducing RTO_{min} at scale to mitigate incast.

Our discussion will proceed as follows. Section 2 will briefly describe our datacenter environment. Section 3 will discuss the three significant problems that we have encountered with TCP in our datacenter. Section 4 will discuss problems that delayed acknowledgements introduce into datacenter networks, and will analyze solutions. Section 5 will discuss reducing RTO_{min} to mitigate incast. Section 6 will discuss addressing the root cause of incast-induced packet loss using DCTCP. Section 7 will discuss obstacles that prevent DCTCP from being used in our environment, and solutions for those problems. Section 8 will then compare DCTCP performance to that of TCP. Section 9 will investigate the performance of automatic TCP buffer tuning in our environment. Section 10 will briefly discuss related work, before we conclude in Section 11.

2 Setting

The majority of recent work on TCP in the datacenter has either implicitly or explicitly been undertaken in the context of an Internet services setting. Of course, datacenter computation applies to a much broader spectrum of applications, and even within a single datacenter of a single organization, a wide variety of application types may be found.

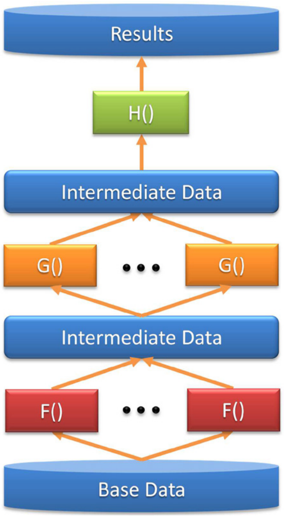


Figure 1: Typical Application Structure

2.1 Overview

The context of this work is a datacenter used largely for two broad types of applications: Monte Carlo simulation and data analysis. A typical application is structured as shown in Figure 1, which shows a common communication-intensive application structure in our datacenter. This application is constructed as a series of transformations (depicted as rectangles) on data (depicted as ovals and rounded rectangles.) Each transformation may read several data elements, and may store several data elements.

Most data access is to one of two highly-parallel distributed data storage systems: a key-value store and a distributed file system. The key-value store tends to generate much higher degrees of incast (discussed at length further in this paper) due to support for bulk reading and writing of values. The distributed file system results in more limited incast as the number of blocks simultaneously read by any particular operation is limited by the file system's read-ahead limit. Further details of these storage systems are outside the scope of this paper, but both are colocated with our computation servers and—thus—are highly parallel.

Monte Carlo simulations tend to be computationally intensive, but even they tend to contain periods of intensive communication. Data analysis applications tend to be storage and communication intensive.

As our datacenter is shared among many applications and distinct user groups, it is very important that applications in our datacenter are as loosely coupled as possible.

Unless otherwise specified, the applications discussed and results presented in this paper were obtained on a 10 Gbps network with a 9K MTU. Also unless otherwise specified, controlled experiments were conducted using iperf as traffic generator sending at the maximum rate allowed. TCP congestion control is CUBIC [6] unless otherwise stated (as CUBIC is the Linux default congestion control.) We conducted several of the controlled experiments using the Linux Reno implementation, but did not observe any significant differences. As such we have left comparisons with Reno (and other TCP variants) as out of scope for this work. Applications in this datacenter do not access the public Internet.

2.2 Traffic Characteristics

To illustrate the type of traffic that our applications generate, we recorded network traffic for a two-minute interval of a representative application (a Monte Carlo simulation) on a single server in this application. Due to the uniform nature of both our applications and our storage systems, the traffic seen by other servers is very similar. (We have verified this with additional samples on other servers.) Figures 2, 3, and 4 summarize flow characteristics of the recorded traffic.

TCP connections in our environment tend to be long-lived. For the purposes of this analysis, we define a flow as packets demarcated by TCP PUSH flags within a single TCP connection.

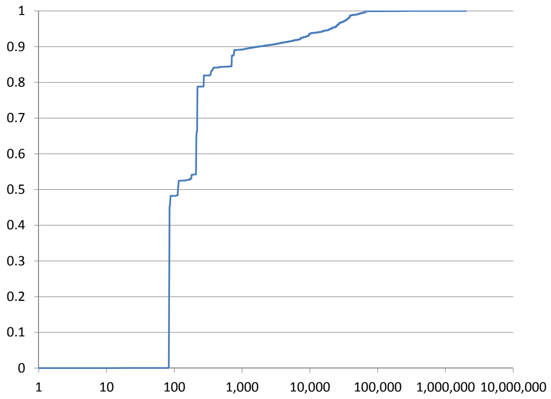


Figure 2: CDF of flow sizes

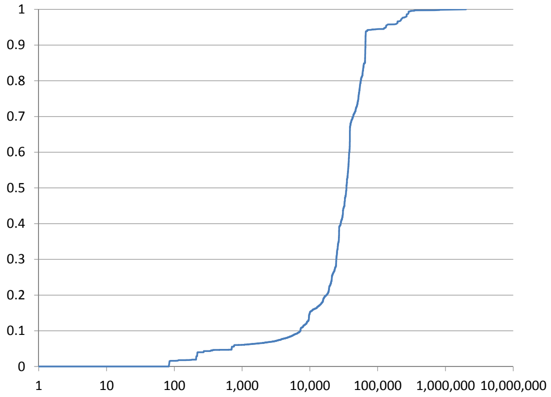


Figure 3: CDF of flow bytes

Figure 2 depicts the cumulative distribution of flow sizes sampled, and illustrates that the vast majority of flows in this application are very small. These small flows largely consist of either data retrieval requests, or simple operation results. (As stated earlier, an individual connection will contain many flows. These flows often occur in quick succession within the connection.)

Figure 3 illustrates the cumulative distribution of flow bytes. (For each flow size, the corresponding point depicts the fraction of total bytes in flows less than or equal to that flow size.) As shown in Figure 3, the majority of bytes are in larger flows—in spite of the large number of small flows. This is due to the fact that while the simple requests and operation results dominate in terms of flow numbers, most bytes on the network are generated by actual value storage or retrieval.

In addition, we also categorized the sampled traffic as shown in Figure 4. This figure shows the fraction of total traffic (measured in bytes) that falls into the given traffic categories. This figure clearly shows that key-value store traffic dominates, followed by distributed file system traffic. Other traffic types are not a significant fraction of the total traffic.

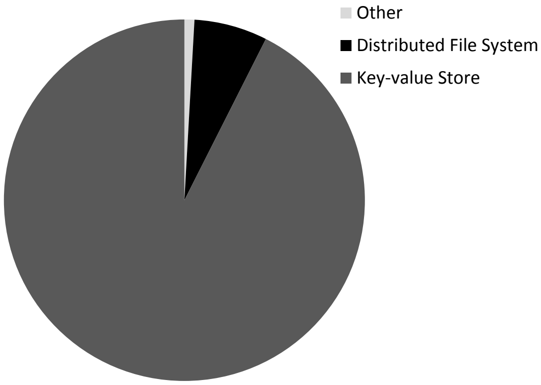


Figure 4: Flow Type Categorization

In summary, key-value store traffic dominates the traffic in the measured application. Most flows generated by this application are very small—too small for traditional congestion control to prevent problems such as incast. The majority of traffic, however, is in contained in larger flows. Thus, congestion control plays an important role in preventing larger flows from experiencing congestion, and in preserving buffer space for small flows.

3 TCP in the Datacenter

Communication intensive datacenter applications present datacenter networks with several performance problems [5]. This is largely due to the fact that TCP—the foundation of many datacenter applications—was not originally designed with the characteristics of modern datacenters in mind. In this section we discuss three significant problems with TCP that we have encountered: delayed ACK induced stalls, incast, and problems with receive buffer tuning.

3.1 Stalls Due to Delayed ACKs

Delayed ACKS in TCP allow TCP to substantially reduce the number of packets sent. Delayed ACKs work by delaying the sending of an ACK for multiple segments. The delayed ACK effectively merges ACKs by cumulatively acknowledging multiple received segments.

Delayed ACKs have an associated timeout to prevent the sender from stalling forever due to a lack of ACKs from the receiver. The default timeout is tens to hundreds of milliseconds. In a datacenter with sub-millisecond RTT, the default delayed ACK timeout is far too large, and we have observed application-level timeouts that were caused by delayed ACKs. Section 4 will discuss resolving this issue.

3.2 Incast

The most vexing problem that TCP encounters in our datacenter network is "TCP incast" [12]. TCP incast occurs whenever a single receiver receives data from multiple senders in a short amount of time. This is a frequent communication pattern in datacenter applications. As depicted in Figure 5, when this situation occurs, the switch to which the receiver is attached is often overloaded: the senders send more data than the receiver can receive; the switch cannot store all of the data; and so the switch discards data that it does not have room for [13]. Unlike delayed ACK-induced timeouts, incast is much more difficult to remedy, and we will spend much of this paper discussing this problem.

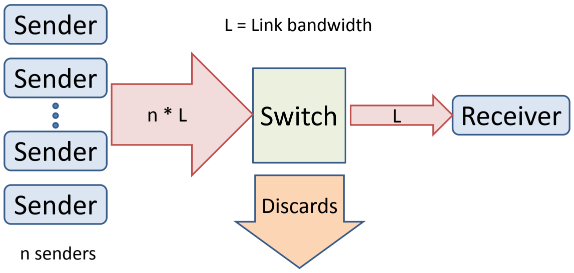


Figure 5: Incast

Previous work discussing incast and other datacenter TCP problems has focused on Internet service applications and shown that TCP performs poorly in datacenters that are servicing these applications. While the nature and structure of our datacenter applications are very different, we still experience similar problems with TCP in our datacenter.

Consider our typical application structure discussed previously and depicted in Figure 1. Each transformation may read several data elements from our distributed storage systems, and may store several data elements into our distributed storage systems. As a result, reads from our distributed storage systems often result in a high degree of TCP incast. Writes to the distributed storage systems also contribute to incast as many writers may be writing to the same storage node.

At a high level, we find that incast produces the following problems at the application layer:

- Communication timeouts and retransmissions
- Lost throughput
- Increased latency
- Latency variance (jitter)

These problems can afflict even "innocent" applications and servers uninvolved in the communication. At the business level, further problems result:

- Application failures
- Idle servers waiting for communication, and increased costs associated with procuring and operating additional servers.
- Application failures even for "innocent bystanders"
- Development effort to work around communication problems
- Effort lost troubleshooting network problems in innocent applications
- Effort lost coordinating among different development groups to avoid communication problems.

3.3 Receive Buffer Tuning

In addition, a very significant problem that we have encountered with TCP in the datacenter is receive buffer tuning [16]. The receive buffer size has a dramatic impact on TCP performance and server RAM utilization.

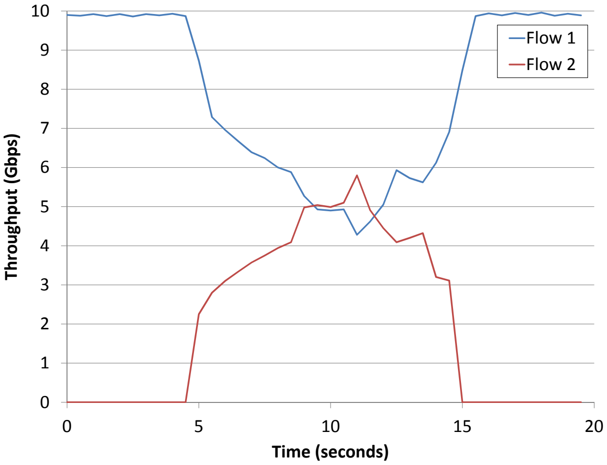


Figure 6: TCP convergence

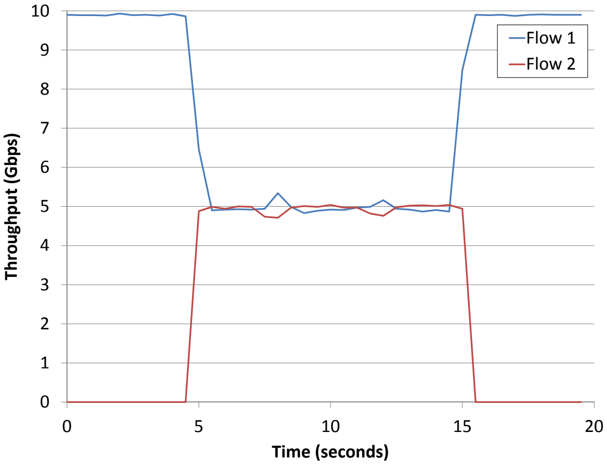


Figure 7: DCTCP convergence

To illustrate this, consider the results of a simple two-flow throughput experiment. Both flows were sent from distinct servers to a common receiver. The first flow ran for 20 seconds. The second flow started 5 seconds later, and ran for a total of 10 seconds. The results are shown in Figure 6.

TCP convergence, fairness, and stability in this test are all extremely poor. TCP should be able to converge within a few RTT, not several seconds. (While [11] discusses some detailed problems with TCP-CUBIC convergence, the behavior shown in Figure 6 is far worse than is expected.)

Figure 7 repeats this test for DCTCP. Surprisingly, while [2] finds that DCTCP converges more slowly than TCP, Figure 7 shows DCTCP dramatically outperforming TCP with respect to stability, convergence, and short-term fairness.

The source of this unexpected behavior is receive buffer tuning. This will be addressed in detail in Section 9.

3.4 Summary

The problems discussed above are significant, and historically we worked around them at the application layer. In the following sections, we discuss how we have largely eliminated these problems, dramatically increased our network performance, and removed the need for application-level workarounds.

4 Delayed ACKs

As discussed earlier, delayed acknowledgements can cause significant problems. Delayed ACK timeouts are—by default—far too large for a datacenter setting. Fortunately, there are two simple alternatives to remedy this problem: 1) eliminate delayed acknowledgements, or 2) reduce the delayed acknowledgement timeout. We have investigated both approaches.

If ACKs could be generated without cost, the ideal ACK delay would be zero, and an ACK would be generated for every single packet. Unfortunately, while eliminating delayed ACKs eliminates the possibility of any sender stall, it does so at the cost of generating a significant number of packets. We do not find this increased load to be a problem in our network, but we do find it to be problematic on our end servers.

Figure 8 illustrates this behavior. In this test, one or two senders send to a single receiver. Delayed ACKs are delayed a maximum of 0 (i.e. no delayed ACKs), 1, or 40 milliseconds. The total CPU % utilized by IRQ daemons on the receiver for the given test is plotted for each test (100% is the equivalent of 1 CPU completely busy). This test exhibits essentially no difference for delays of 1 and 40 milliseconds. Turning delayed acknowledgements entirely off, however, produces a sharp increase in CPU utilization for both one and two flows. (Repeating this test yields similar results with insignificant variation.)

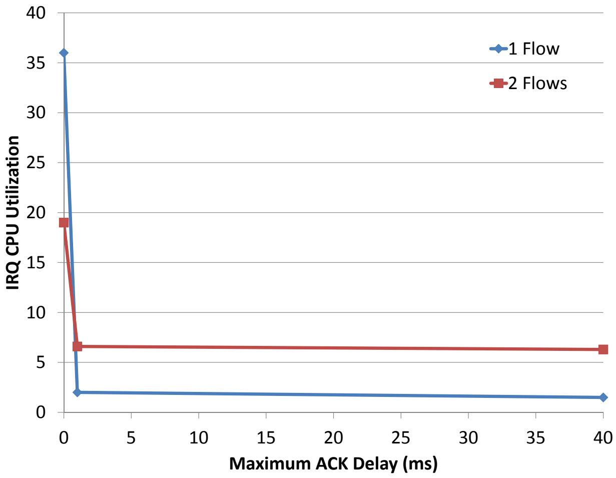


Figure 8: Delayed ACK CPU Utilization

For this reason, in our network we now lower the delayed ACK timeout as much as possible without turning delayed ACKS off. Those constraints yield a delayed ACK setting of 1 ms. We will still incur an occasional stall, but the stall is not long enough to cause significant issues at our application layer. (Some applications, however, may benefit from turning off delayed ACKs entirely.)

5 Reducing RTO_{min}

During the most communication-intensive phases of our application, we found that our applications were experiencing large numbers of incast-induced TCP timeouts. At the application layer, this resulted in a long tail on our task completion times. The effects of incast are clearly seen in Figure 9 which shows a TCP sequence graph from a single flow of a production application during a heavy all-to-all incast. The duplicate sequence numbers visible are packet losses and retransmissions that were successfully handled by TCP. The 200 ms pauses in the flow, however, are due to whole-window loss induced TCP timeouts incuring the RTO_{min} penalty.

Previous work [13] has proposed a simple technique to mitigate the effects of incast-induced TCP timeouts: reduce RTO_{min}. We employed this technique in our datacenter, and the benefits can be seen in Figure 10 which shows TCP sequence plot of a flow experiencing incast. As with Figure 9, loss is visible, as is a timeout, but timeouts are reduced to 5 ms which is the minimum effective RTO_{min} that our servers support.

As shown in Figures 9 and 10, reducing RTO_{min} significantly improved the performance of TCP in our datacenter by mitigating the effect of TCP timeouts. It did

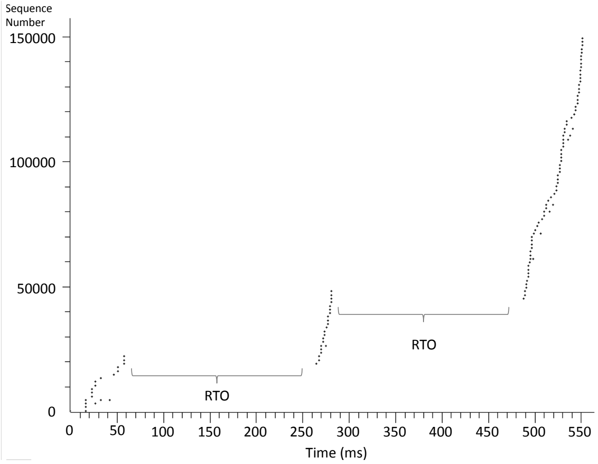


Figure 9: RTO_{min} 200 ms

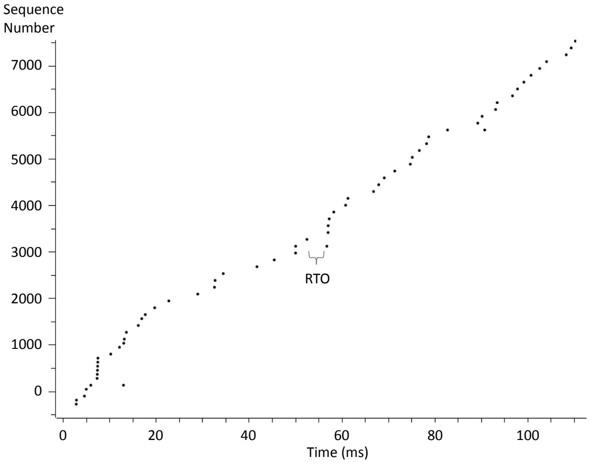


Figure 10: RTO_{min} 5 ms

not, however, prevent timeouts. In fact, the rate of packet loss in our network *increased* significantly after we applied the RTO_{min} change. This is expected as lowering the RTO_{min} does not prevent packet loss and timeouts, it just mitigates the effects. Moreover, lower timeout values will increase the number of contending flows which will tend to increase the overall number of lost packets.

In short, we found that reducing RTO_{min} greatly reduced the impact of incast on our applications. Network and server stability were not impacted by this change even when running on a cluster of over 2,000 servers. Innocent applications (applications not involved in the incast) were, however, still impacted. Moreover, network performance was still far from ideal. We were still incurring (much smaller) timeouts and the usual TCP latency. In the next section, we discuss addressing the root cause of incast.

6 DCTCP

Subsequent work on datacenter TCP has proposed several techniques to actually reduce packet losses due to incast, rather than just mitigate the effects of lost packets. Of these techniques, one of the most promising for deployment in our datacenter is DCTCP. DCTCP possesses several features that make it a particularly promising approach for us: it relies on capabilities that are available in current hardware and software, an implementation is available [8], and it does not contain features that we cannot use. (In particular, we decided against leveraging work that relies on flow priority or deadlines as our connections are long-lived and utilized for many different types of communication. As a result, communicating priority or deadline information to the network layer would be difficult or impossible for our applications.)

Our primary objectives for moving to DCTCP were to: eliminate TCP timeouts (or nearly eliminate them), reduce latency, and reduce the network-induced coupling of applications. In particular, we wanted to protect "innocent bystanders" from aggressive applications.

In the following sections, we first discuss obstacles to reaping these benefits, and how we extended DCTCP to overcome these obstacles, followed by some discussion of our extended DCTCP's performance.

7 DCTCP Deployment Challenges

7.1 Coexistence with TCP

In motivating the design of DCTCP, [1] states "[a datacenter] network is largely homogeneous and under a single administrative control. Thus backward compatibility, incremental deployment and fairness to legacy protocols are not major concerns." For actual usage in our datacenter, however, these are *all* major concerns. We do not have the luxury of a "big bang" deployment for several reasons.

- There are multiple applications running in our datacenter with distinct ownership. It is critical that one application moving to DCTCP does not negatively impact any application using conventional TCP. Recall that one of our major arguments for deploying DCTCP is to *reduce* the coupling of applications.
- Many critical services cannot be moved to DCTCP. Even for applications with owners willing to make the move to DCTCP, there are services used by those applications that we simply cannot move to DCTCP. For instance, many of our applications leverage file servers that do not support DCTCP.

Unfortunately, DCTCP and TCP do not naturally coexist well. To demonstrate this, we conducted a simple test where one TCP flow and one DCTCP flow both send at maximum rate from distinct servers to a single receiver. (Again, these experiments are conducted on a 10 Gbps network using iperf as sender and receiver.) The TCP flow lasts for a total of 20 seconds. The DCTCP flow lasts for 10 seconds and starts 5 seconds *after* the TCP flow starts.

The results are shown in Figure 11. As soon as the DCTCP flow starts, the TCP flow almost completely stops while the DCTCP flow completely saturates the link. This is an extremely negative result, and essentially the complete opposite of what we require. (Note that it is possible to delay the onset of this behavior through configuration settings, but this will not solve the fundamental problem.)

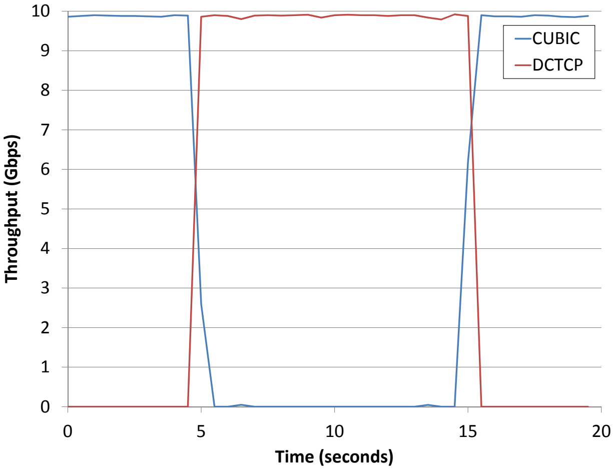


Figure 11: DCTCP Coexistence with TCP

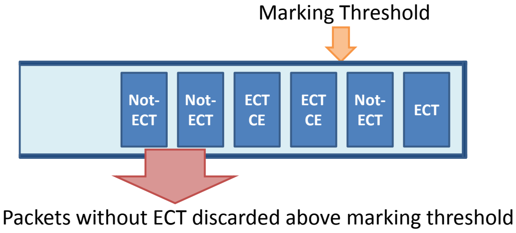


Figure 12: Switch RED ECN Implementation

The reason that DCTCP traffic dominates conventional TCP traffic is due to RED/ECN AQM behavior which is as follows for a switch configured for DCTCP. As depicted in Figure 12, when the switch queue length is below the marking threshold (there is only one threshold for DCTCP), any packet that arrives is simply queued irrespective of ECT status. When the queue length is over the marking threshold, however, all ECT packets are marked with CE, but non-ECT packets are *dropped*. In DCTCP, the marking threshold is set very low value to reduce queueing delay, thus a relatively small amount of congestion will exceed the marking threshold. During such periods of congestion, conventional TCP will suffer packet losses and quickly scale back *cwnd*. DCTCP, on the other hand, will use the fraction of marked packets to scale back *cwnd*. Only when all packets are marked will *cwnd* be scaled back as far as conventional TCP. Thus rate reduction in DCTCP will be much lower than that of conventional TCP, and DCTCP traffic will dominate conventional TCP traffic traversing the same link.

As both TCP and DCTCP must service the same servers in our network, we resort to utilizing IP DSCP bits to segregate DCTCP traffic from conventional TCP traffic. AQM is applied to DCTCP traffic, while TCP traffic is managed via drop-tail queueing.

7.2 Non-compliant switches

While we are fortunate enough to have support for ECN marking on our top-of-rack switches, this is the only location in our network that supports ECN marking. Higher-level switches are purely drop-tail. DCTCP must gracefully support transit over non-ECN switches without impacting either the behavior of DCTCP traffic or conventional traffic. Our tests show that DCTCP successfully resorts to loss-based congestion control when transiting a congested drop-tail link.

7.3 Non-technical challenges

Even without any technical challenges, altering the network in a major enterprise is a difficult undertaking. Network administrators are, necessarily, risk-averse. A reliable network is a business-critical requirement. Thus, network innovations are often viewed as presenting significantly more risk than reward.

We were able to present a compelling case for DCTCP implementation due to the following:

- Reduction in coupling. Application coupling was a known phenomenon in our datacenter. Conventional TCP's strong coupling of unrelated applications causes problems as discussed previously. DCTCP's promise to greatly reduce the coupling between applications meant that our network administrators would directly benefit from reduced troubleshooting requests from applications experiencing mysterious network performance issues caused by unrelated applications.
- Timing. We timed our DCTCP roll-out to coincide with the deployment of new network switches in our environment. We worked with our network administrators to ensure that the switch features necessary to support DCTCP were available from day one.
- Primum non nocere. Our support for conventional TCP and non-ECN compliant switches enabled us to guarantee that we would not harm existing applications.

7.4 Connection Establishment

Segregating DCTCP from conventional TCP removed one potential showstopper from our DCTCP deployment effort. Nevertheless, we encountered one other major problem in DCTCP that had the potential to prevent DCTCP adoption in our network: we found that under load, DCTCP would fail to establish network connections due to a lack of ECT in SYN and SYN-ACK packets.

[1] does not discuss setting ECT on SYN and SYN-ACK packets. The Stanford implementation [8] does *not* set ECT on either SYN or SYN-ACK packets. This is in line with RFC 3168 [14] which states *"A host MUST NOT set ECT on SYN or SYN-ACK packets."* RFC 5562 [10] (derived from ECN+ [9]) proposes setting ECT on SYN-ACK packets, but maintains the restriction of no ECT on SYN packets.

RFC 3168 and RFC 5562 prohibit ECT in SYN packets due to security concerns regarding malicious SYN packets with ECT set. These RFCs, however, are intended for general Internet use, and do not directly apply to DCTCP. In our internal network, we do not tolerate the compromised servers necessary for an attacker to send such packets. Moreover, the Stanford implementation's adoption of these RFCs likely owes more to its leveraging of the existing ECN support in Linux than anything else.

We find that setting ECT on SYN and SYN-ACK is critical for the practical deployment of DCTCP. Without this feature, SYN and SYN-ACK packets will be dropped whenever there is even minor congestion. As discussed in Section 7.1, and depicted in Figure 12, whenever the queue length is greater than the marking threshold, non-ECT packets are dropped. Thus, if SYN and SYN-ACK packets are non-ECT they will be dropped with high probability. We modified DCTCP to apply ECT to both SYN and SYN-ACK packets. We refer to this implementation as "DCTCP+" to distinguish it from the original DCTCP implementation. (Following the naming convention of ECN+ which extended ECN with ECT on SYN-ACK only.)

To measure the effect of this issue, we conducted an experiment where we disabled ECT for SYN packets and attempted to establish a DCTCP connection (with no SYN or SYN-ACK ECT) in the presence of a number of competing DCTCP+ flows which were already established and sending data at maximum rate. As shown in Figure 13, as the number of competing flows increases, it quickly becomes hard, then impossible, to establish a connection when SYN packets are non-ECT. Thus, we utilize DCTCP+ in our deployment, which marks both SYN and SYN-ACK packets as ECT.

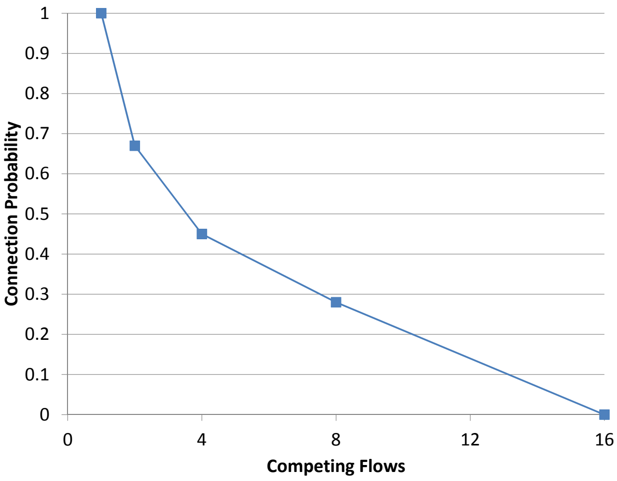


Figure 13: Connection Probability without SYN ECT

Note that given our support for conventional TCP, we could use DSCP to cause SYN and SYN-ACK packets *only* to be treated as conventional TCP. We do not take this approach as it would split packets from a single flow across two separate paths in our network which is highly undesirable.

8 DCTCP+ Performance

We now discuss several elements of DCTCP+ performance illustrating where DCTCP+ does well, and where there is room for improvement.

8.1 Incast Throughput and Fairness with Buffer Tuning Active

We first measured performance in an incast scenario similar to that in Figure 5. In this case, a single receiver received traffic from 19 senders for a total of 10 seconds (as discussed previously all experiments in this section are conducted on a 10 Gbps network). Importantly, automatic receive buffer tuning is *on* for this test; we will later show that this has a dramatic effect on TCP performance but very little for DCTCP+. Figures 14 and 15 show summarized throughput statistics for all 19 flows for each experiment. DCTCP+ fairly distributes the link bandwidth among flows resulting in a very narrow throughput distribution while fully utilizing the link. TCP is also able to fully utilize the link, but does so very inefficiently as flows stall due to a combination of packet loss and incorrectly sized receive buffers. The link is able to remain utilized, however, as other flows step in and utilize the missing bandwidth. Nevertheless, the median throughput is lower, and there is a large variation among flow throughput. In short, under DCTCP+, flow performance is fast and reliable while under TCP, packet loss and the poor performance of buffer auto tuning causes extremely variable throughput.

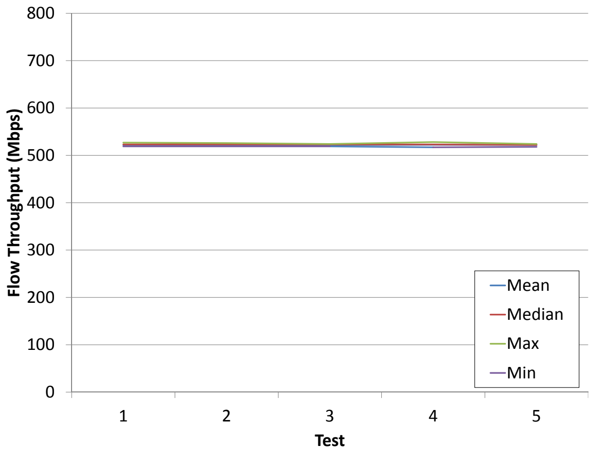


Figure 14: DCTCP single-receiver incast

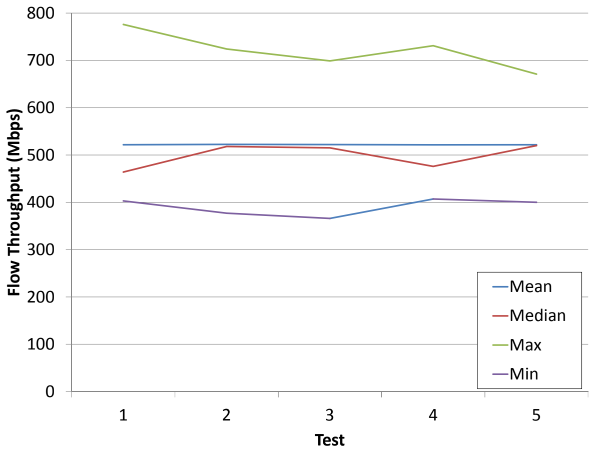


Figure 15: TCP single-receiver incast (buffer tuning active)

	TCP	DCTCP+
Mean	4.01	0.0422
Median	4.06	0.0395
Maximum	4.20	0.0850
Minimum	3.32	0.0280
σ	0.167	0.0106

Table 1: Per-packet latency in ms

Moreover, as shown in Table 1, per-packet latency under TCP is two orders of magnitude greater than per-packet latency under DCTCP+. DCTCP+'s reliably low latency enables higher-layer applications to reliably communicate in a very short time span. Under TCP (particularly before we lowered RTO_{min}), our applications needed added logic to deal with the unpredictable latency and throughput that incast induced. DCTCP+'s consistently superb performance make it a superior transport protocol to TCP within our datacenter.

8.2 Scale

The scalability afforded by datacenter computing lies at the heart of applications ranging from web search engines, to the Monte Carlo simulations and data analytics running in our datacenter. Realizing the benefits of scale, however, is challenging for many components of networked systems. For DCTCP+ to be an effective datacenter transport mechanism, it must scale with the applications that it supports.

In this section, we examine the scalability of DCTCP+ experimentally and analytically.

Incast traffic patterns are particularly difficult to scale. We examined DCTCP+ support at scale for incast by sending large numbers of long-lived flows (20 seconds) from many senders to a single receiver. Each flow was generated by a distinct server using iperf. Ideally, we should see that—as with TCP—the link would be fully utilized and each flow would receive a fair share of the link, but—unlike TCP—latency would remain low.

The results of this test are shown in Tables 2 and 3. Table 2 shows that throughput and long-term fairness are excellent through 500 servers. Table 3, however, exhibits some problems. The first problem to notice is that latency is relatively high even for 100 servers. At 300 servers, the high latency shows that the receive queue is entirely full, and at 400 servers significant amounts of traffic are lost and numerous timeouts are occurring. By 500 servers, 8.7% of packets sent are retransmissions, and timeouts are very significant; as a result, short term flow fairness will be poor. In a nutshell, it seems that at this scale, DCTCP+ is performing no better than TCP.

Senders	Total	Mean	Max	Min	σ
100	9,901	99.0	99.3	88.6	1.06
200	9,900	49.7	49.9	46.1	0.35
300	9,901	33.2	34	31.2	0.36
400	9,894	24.9	28.5	20.2	1.01
500	9,895	20.0	23.9	13.8	1.42

Table 2: Scale Test: Throughput (Mbps)

		Retransmissions		
Senders	RTT (ms)	Total	%	RTO
100	1.60	0	0	0
200	3.11	0	0	0
300	4.38	3	0	0
400	4.42	702	4.6	274
500	4.44	1110	8.7	655

Table 3: Scale Test: Latency and Retransmissions

Why is latency so high? Shouldn't the switch be marking packets causing DCTCP+ to back off before latency gets so high? Packet traces from a sender involved in this test show that for all cases, the switch is marking 100% of packets in steady state, yet DCTCP+ is still sending packets. In other words, even when the switch is telling DCTCP+ to fall back aggressively, DCTCP+ refuses to fall back enough to prevent congestion.

The source of this behavior is in the *cwnd* update procedure of DCTCP+. According to [1], DCTCP+ updates *cwnd* as:

$$cwnd \leftarrow cwnd \times (1 - \alpha/2)$$

Actual TCP implementations, however, are more intricate, and the Linux implementation in [8] updates *cwnd* as follows:

```
cwnd_new = max(tp->snd_cwnd
               - ((tp->snd_cwnd
                   * tp->dctcp_alpha)>>11),
               2U);
```

In other words, irrespective of measured congestion, DCTCP+ will always be willing to send two segments. This effectively puts a lower limit on DCTCP+ transmission rate per sender of:

$$TransmissionRate \geq \frac{SegmentSize \times 2}{RTT}$$

The resulting load for our scale test is shown in Table 4.

Senders	Load (Gbps)
100	3.27
200	6.55
300	9.82
400	13.09
500	16.36

Table 4: DCTCP+ Load vs. Scale

By 300 servers, load is nearly at the capacity of the link, and at higher scales, the load exceeds the link capacity. The result is the significant packet drops, retransmissions, and timeouts shown above. In effect, once the load due to the DCTCP+ minimum transmission rate exceeds the link capacity, DCTCP+ congestion control is no longer in effect, and TCP congestion control takes over. Hence, at scales higher than 300 in this test, DCTCP+ congestion control is no longer in effect.

DCTCP+ scale can be extended by reducing the minimum transmission rate per server. This can be done by applying the *cwnd* cap logic found elsewhere in the Linux TCP implementation.

```
cwnd_new = max(tp->snd_cwnd
               - ((tp->snd_cwnd
                   * tp->dctcp_alpha)>>11),
               1U);
cwnd_new = min(cwnd_new,
               tcp_packets_in_flight(tp) + 1U);
```

With this addition, under a congested network, only one packet will be allowed per RTT and the scaling will double – just over 600 servers can send at full rate to a single receiver without the minimum DCTCP+ transmission rate exceeding the link capacity.

While this change may result in additional delayed acknowledgements, our initial evaluation indicates that lowering the delayed acknowledgement timeout as discussed in Section 4 mitigates this concern. We leave a full evaluation for future work.

8.3 Operational Experience

We have been running DCTCP+ at a scale of approximately 600 servers for nearly one and a half years as of this writing. While quantifying the isolated benefits of DCTCP+ is ongoing work, qualitatively, we have found DCTCP+ to be a stable transport protocol and with the RTO_{min} reduction, delayed ACK reduction, and DCTCP+ all in place, we no longer observe any application-layer issues that are caused by TCP. This is a significant improvement.

9 Receive Buffer Tuning

Figures 6 and 7 previously showed an unexpected result: TCP converging more slowly than DCTCP, and generally performing very poorly. Careful analysis of Figures 17a&b in [1] shows that the creators of DCTCP observed similar behavior experimentally (though this particular behavior was not discussed in [1]): DCTCP outperforms TCP in their experiment with respect to stability, convergence, and short-term fairness. We repeat this convergence experiment in our network under several scenarios—first on a 1 Gbps network, then on a 10 Gbps network. The 1 Gbps result for TCP is shown in Figure 16. This closely matches the results from [1].

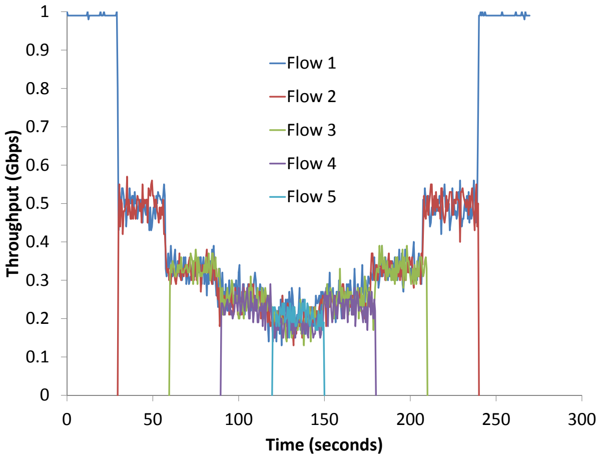


Figure 16: 1G, TCP, Buffer Tuning On

Figure 17 shows that moving to a 10 Gbps network exacerbates the problems with convergence, fairness and stability. We find that, as with the 1 Gbps result presented in [1], at 10 Gbps DCTCP+ convergence is superior to TCP convergence as shown in Figure 18.

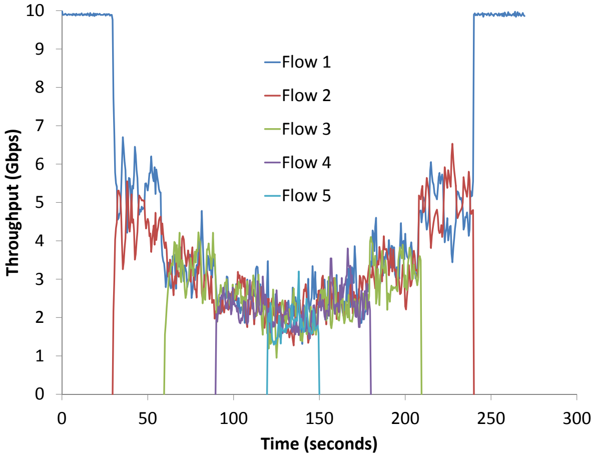


Figure 17: TCP, Buffer Tuning On

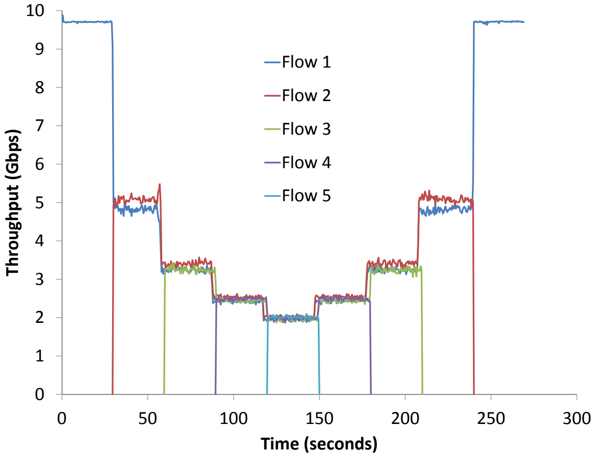


Figure 18: DCTCP+, Buffer Tuning On

These results seemingly defy [2] which showed DCTCP converging more slowly than TCP. The cause of this problem has a simple explanation: receive buffer tuning. Historically, network developers were tasked with setting TCP buffer sizes manually. Getting the buffer sizes right is important for both network and end-system performance: undersized buffers hurt network throughput; overly generous buffer sizes consume RAM, impact system performance, and limit application scale. It is possible to manually set buffer sizes to attempt to strike a balance, but this is very undesirable as it binds the performance of an application to the behavior of a particular network. Moreover, it fails to allow dynamic memory management to take into account a server's memory state.

To overcome these limitations, several approaches have been developed to dynamically set TCP buffer sizes. Unfortunately, in a datacenter setting, these algorithms can perform poorly. In principle, the receive buffer of an application should be set to the bandwidth delay product (BDP) of a link. The trouble is that inside of a datacenter, propagation delay is extremely small—approximately four orders of magnitude less than the queueing delay of a congested link! As a result, the bandwidth delay product of a link varies significantly, and—worse—is a function of the receive buffer size. The strong feedback present in receive buffer tuning a TCP link makes tuning a difficult problem. The tuning algorithm takes many seconds to adapt from the low-latency congestion-free regime to the high latency congested regime. As a result, TCP performance in our datacenter is very poor when automatic receive buffer tuning is enabled. This also explains why DCTCP+ is able to outperform TCP: DCTCP+ keeps latency far lower than TCP. As a result, the tuning algorithm experiences far less feedback and has a much easier time finding the correct buffer size.

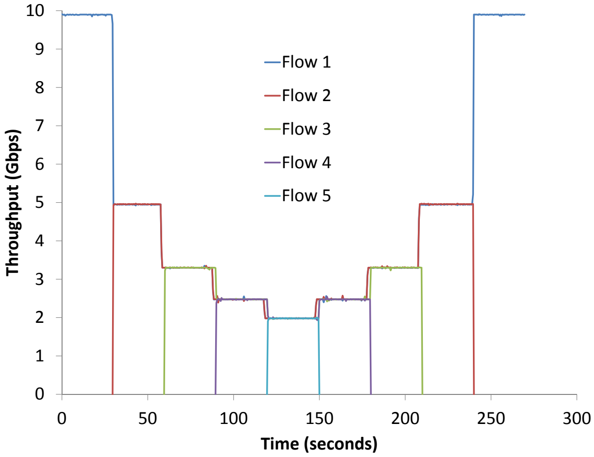


Figure 19: TCP, Buffer Tuning Off

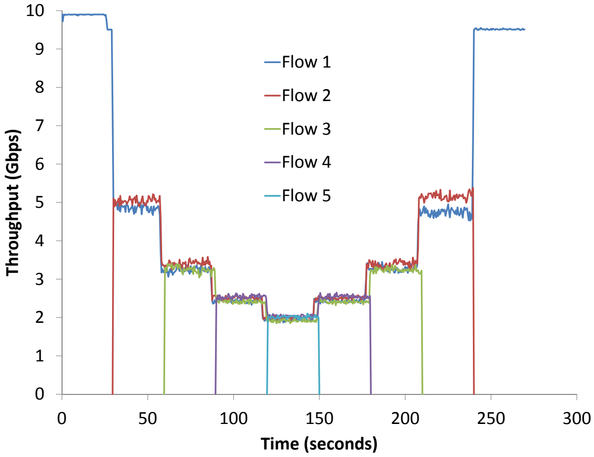


Figure 20: DCTCP+, Buffer Tuning Off

Turning off receive buffer tuning, and manually setting the receive buffer size to be greater than the maximum delay bandwidth product possible, results in much better behavior for TCP, as shown in Figure 19. With this change, TCP stability, convergence, and fairness all exceed that of DCTCP+. DCTCP+ performance, on the other hand, is not changed significantly by manually setting the buffer size, as shown in Figure 20.

In summary, receive buffer tuning can have a dramatic impact on TCP performance. The anomalous results shown in Figures 17a&b of [1], and discussed in this paper, are explained by poor tuning of the TCP receive buffer. With proper receive buffer sizing, TCP stability, convergence, and fairness outperform DCTCP+. Achieving proper receive buffer sizing, however, is much more difficult under TCP than DCTCP+ due to the massive dynamic range of latencies that even two competing flows can generate.

10 Related Work

TCP incast was first discussed by Nagle et al. [12]. Phanishayee et al. [13] explored solutions such as reducing RTO_{min}. Vasudevan et al. [17] proposed reducing RTO_{min} further using fine-grained timers. Instead of this, we simply reduced RTO_{min} as far as our kernel was capable of.

Yu et al. [20] analyze application performance in the datacenter network of an Internet service provider; they identify several performance problems caused by applications, the end-server network stacks, and the network itself. We independently have encountered similar problems in a completely different context, and we believe that the problems encountered in [20] are general problems likely to be found widely in datacenter communication. To fix the problems with delayed acknowledgements, [20] suggests either reducing the delayed ack timeouts or disabling delayed acks. Our work goes further by analyzing the tradeoff between these two options.

Wu et al. [19] also observe that switches running RED/ECN drop non-ECT packets, but do not discuss the impact of this behavior on DCTCP.

Semke et al. [16] developed a method of automatically tuning TCP buffers that is the basis of the current Linux autotuning algorithm.

There has been a good deal of work—such as [7] [18] [21]—on achieving superior congestion control than that attainable by DCTCP by incorporating knowledge of flow priorities and deadlines into congestion control. Unfortunately, these techniques are not readily applicable in our environment.

pFrabric [3] takes a clean-slate approach to datacenter communication. This is promising work, but outside of the scope of our work as we were restricted to techniques that we could run in production today.

11 Conclusion

TCP has been tremendously successful in the Internet, and is a ubiquitous protocol that is critical to countless applications. TCP support in datacenters promises to allow these applications to run alongside new applications. Unfortunately, however, experience has shown that TCP's design assumptions break down inside modern datacenters, and performance is often inadequate.

In this paper, we have shown that leveraging recent work overcomes the major deficiencies of TCP inside of the datacenter. We have shown that DCTCP coexistence with TCP is critical in our environment, and demonstrated how this can be accomplished. Moreover, we have shown how a small extension to DCTCP—employing ECT in SYN and SYN-ACK packets—removes a potentially fatal problem with DCTCP.

Nevertheless, this work has also highlighted areas for future work. Despite the dramatic impact on performance that it can have in current implementations, receive buffer auto tuning can perform very poorly. In addition, we have shown how DCTCP scale can be improved; ideally DCTCP would scale even further before filling queues and reverting to TCP.

In closing, deploying recently developed improvements to TCP (along with our extensions) has dramatically improved TCP performance in our datacenter, without requiring any modifications to our applications or distributed storage systems.

Acknowledgements

We would like to thank our reviewers for their feedback. In addition, we greatly appreciate the assistance of Daniel Varga, Wai-Hong Lui, Vikas Chawla, Eric Hagberg, John MacIntyre, Peter King, Johnathan Edwards, Sam Shteingart, Michael Jansen, Jason Greenberg, Matthew Whitehead, Daniel Borkmann, Florian Westphal, and Frank Hirtz,

References

[1] M. Alizadeh, A. Greenberg, D. A. Maltz, J. Padhye, P. Patel, B. Prabhakar, S. Sengupta, and M. Sridharan. Data center tcp (dctcp). In *Proceedings of SIGCOMM 2010*, 2010.

[2] M. Alizadeh, A. Javanmard, and B. Prabhakar. Analysis of dctcp: Stability, convergence, and fairness. In *Proceedings of SIGMETRICS 2011*, 2011.

[3] M. Alizadeh, S. Yang, M. Sharif, S. Katti, N. McKeown, B. Prabhakar, and S. Shenker. pfabric: Minimal near-optimal datacenter transport. In *Proceedings of SIGCOMM 2013*, 2013.

[4] L. A. Barroso and U. Hlzle. *The Datacenter as a Computer: An Introduction to the Design of Warehouse-Scale Machines*. Morgan and Claypool Publishers, 2009.

[5] P. Gill, N. Jain, and N. Nagappan. Understanding network failures in data centers: measurement,

analysis, and implications. In *Proceedings of SIGCOMM 2011*, 2011.

[6] S. Ha, I. Rhee, and L. Xu. Cubic: a new tcp-friendly high-speed tcp variant. *ACM SIGOPS Operating Systems Review*, 2008.

[7] C.-Y. Hong, M. Caesar, and P. Godfrey. Finishing flows quickly with preemptive scheduling. In *Proceedings of SIGCOMM 2012*, 2012.

[8] A. Kabbani, M. Yasuda, and M. Alizadeh. Dctcp-linux. In *https://github.com/myasuda/DCTCP-Linux*, 2012.

[9] A. Kuzmanovic. The power of explicit congestion notification. In *Proceedings of SIGCOMM 2005*, 2005.

[10] A. Kuzmanovic, A. Mondal, S. Floyd, and K. Ramakrishnan. Adding explicit congestion notification (ecn) capability to tcp's syn/ack packets. In *RFC 5562*, 2009.

[11] D. Leith, R. Shorten, and G. McCullagh. Experimental evaluation of cubic-tcp. In *Proceedings of PFLDnet 2008*, 2008.

[12] D. Nagle, D. Serenyi, and A. Matthews. The panasas activescale storage cluster - delivering scalable high bandwidth storage. In *Proceedings of Supercomputing 2004*, 2004.

[13] 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 *Proceeding of FAST 2008*, 2008.

[14] K. Ramakrishnan, S. Floyd, and D. Black. The addition of explicit congestion notification (ecn) to ip. In *RFC 3168*, 2001.

[15] J. Rothschild. High performance at massive scale: Lessons learned at facebook. In *mms://video-jsoe.ucsd.edu/calit2/JeffRothschildFacebook.wmv*.

[16] J. Semke, J. Mahdavi, and M. Mathis. Automatic tcp buffer tuning. In *Proceedings of SIGCOMM 1998*, 1998.

[17] V. Vasudevan, A. hanishayee, H. Shah, E. Krevat, D. Andersen, G. Ganger, G. Gibson, and B. Mueller. Safe and effective fine-grained tcp retransmissions for datacenter communication. In *Proceedings of SIGCOMM 2010*, 2010.

[18] C. Wilson, H. Ballani, T. Karagiannis, and A. Rowtron. Better never than late: Meeting deadlines in datacenter networks. In *ACM SIGCOMM Computer Communication Review*, 2011.

[19] H. Wu, J. Ju, G. Lu, C. Guo, Y. Xiong, and Y. Zhang. Tuning ecn for data center networks. In *Proceedings of CoNEXT 2012*, 2012.

[20] M. Yu, A. Greenberg, D. Maltz, J. Rexford, L. Yuan, S. Kandula, and C. Kim. Profiling network performance for multi-tier data center applications. In *Proceedings of NSDI 2011*, 2011.

[21] D. Zats, T. Das, P. Mohan, D. Borthakur, and R. Katz. Detail: reducing the flow completion time tail in datacenter networks. In *Proceedings of SIGCOMM 2012*, 2012.

Beyond Sensing: Multi-GHz Realtime Spectrum Analytics

Lixin Shi* Paramvir Bahl† Dina Katabi*
*Massachusetts Institute of Technology, {lixin, dina}@csail.mit.edu
† Microsoft Research, Redmond, WA, bahl@microsoft.com

Abstract – Spectrum sensing has been an active research area for the past two decades. Nonetheless, current spectrum sensing systems provide only coarse occupancy data. They lack information about the detailed signal patterns in each band and can easily miss fleeting signals like radar.

This paper presents SpecInsight, a system for acquiring a detailed view of 4 GHz of spectrum in realtime. SpecInsight's design addresses the intrinsic conflict between the need to quickly scan a wide spectrum and the desire to obtain very detailed information about each band. Its key enabler is a learned database of signal patterns and a new scheduling algorithm that leverages these patterns to identify when to sample each band to maximize the probability of sensing active signals.

SpecInsight is implemented using off-the-shelf USRP radios with only tens of MHz of instantaneous bandwidth, but it is able to sense 4 GHz of spectrum, and capture very low duty-cycle signals in the radar band. Using SpecInsight, we perform a large-scale study of the spectrum in 7 locations in the US that span major cities and suburban areas, and build a first-of-its-kind database of spectrum usage patterns.

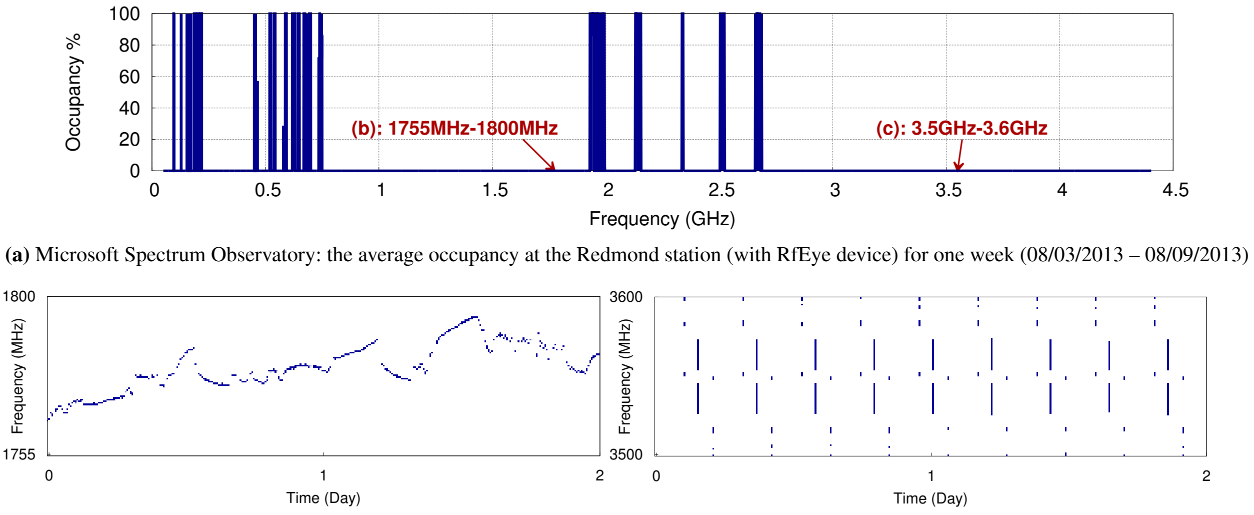
1 Introduction

There has been a significant interest over the past two decades in sensing the wireless spectrum and understanding how it is used [32, 34, 16]. Spectrum sensing has been a recurring topic not only for the research community [6, 26], but also for the government [29, 9], the military [1], and industry [20, 21]. Despite all of these efforts, our understanding of the wireless spectrum is still quite limited. State-of-the-art sensing equipment provide only coarse information of spectrum occupancy. Consider for example the Microsoft Spectrum Observatory (MSO), a state-of-the-art large-scale system for tracking spectrum usage [21]. Fig. 1(a) shows a typical MSO spectrum report. The figure reveals important information about spectrum occupancy, over a span of multiple GHz. Yet, the figure also misses informative details about how the spectrum is used. If one focuses the sensing resources on a single band and continuously listens to that band, one would discover that the above report has missed the fleeting (low duty-cycle) signal in the radar band around 3.5 to 3.6 GHz, which is shown in Fig. 1(c). In fact, not only did it miss the presence of the signal but it also missed how the signal uses the spectrum – i.e., its periodicity in time and its span in frequency. There are many signals that are missed in the MSO report. Fig. 1(b) shows another example. The band is used by the Air Force Satellite Control Network. The signal in the figure is difficult to catch since it hops in a 45 MHz band, occupying only 1 kHz at a time, i.e., its occupancy is 2×10^{-5}.

Learning the details of how the spectrum is used – e.g., the time-frequency utilization patterns in Fig. 1(b) and Fig. 1(c) – is fundamental to the design of dynamic spectrum access (DSA) systems as it can significantly increase the opportunity for spectrum sharing by leveraging signal periodicity. A band that has a periodic occupancy like the one in Fig. 1(c) can be easily time multiplexed with secondary users. The information can also reveal breaches of spectrum regulations by detecting abnormal utilization patterns, which would be invisible in coarse occupancy reports. The utilization patterns could also provide insight into the diverse technologies occupying the spectrum. The research community may know the technologies in the ISM and Cellular bands. Yet, the vast majority of the spectrum is occupied by undocumented technologies (e.g., radios in government bands), which are little known to the research community.

However, obtaining detailed spectrum utilization patterns is challenging, particularly for low occupancy signals like those in Fig. 1. Sensing hardware has limited bandwidth and cannot acquire multiple GHz in realtime. Therefore, spectrum sensing platforms like those used by Microsoft resort to sequential scanning of the spectrum; they hop from one band to the next, sensing only tens of MHz at any moment [21]. As a result, they obtain only high level occupancy statistics; but they can neither detect the low-occupancy signals nor identify their utilization patterns. Scaling the sensing system to a GHz-wide bandwidth, while obtaining fine-grained information about each band, is a significant challenge that remains unaddressed by past work.

This paper introduces SpecInsight, a multi-GHz spectrum sensing system that reveals the detailed patterns of spectrum utilization in real-time. Underlying our design



(a) Microsoft Spectrum Observatory: the average occupancy at the Redmond station (with RfEye device) for one week (08/03/2013 – 08/09/2013)

(b) Army and Air Force Band (1755 MHz-1800 MHz)

(c) Radar Band (3.5GHz-3.6GHz)

Figure 1: Occupancy vs. Realtime Spectrum Patterns: The top graph shows an occupancy report obtained by the Microsoft Spectrum Observatory (MSO). Today's sensing reports can easily miss low occupancy signals. For example, the report in (a) has missed the Air Force Signal in (b) and the radar signal in (c). Graphs(b&c) are examples of SpecInsight's output, which captures the spectrum time-frequency patterns. The patterns are visualized as intensity maps, where the vertical and horizontal axes represent frequency and time respectively.

is a basic insight that any sensing system using a commodity radio is limited to tens of MHz at a time, and hence will have to sample the multi-GHz spectrum. The question, however, is: Which bands should we sample at what times in order to minimize the probability of missing active signals?

We address this question by observing that many spectrum bands are used according to some time-frequency patterns (e.g., always-on in time and frequency, always-on but hopping periodically in frequency, periodic in time but fixed in frequency, etc.). By learning these patterns, SpecInsight can schedule its scans of the various spectrum bands so as to maximize the probability that it will detect the presence, absence, and variation of spectrum utilization patterns, in every band.

SpecInsight implements this design principle in two phases. First, SpecInsight has an innovative algorithm for learning spectrum utilization patterns. In contrast to past work on detecting WiFi or other technologies in the ISM band, our algorithm has to search for previously unknown patterns without making assumptions about the technologies occupying a particular band. The output of the algorithm is used to populate a database of spectrum patterns and their locations. Second, SpecInsight has a smart scheduling algorithm that leverages the spectrum patterns in the database to sense multiple GHz using only tens of MHz of bandwidth, and still output the detailed spectrum utilization patterns as they occur in real-time. The algorithm is formalized as a multi-armed bandit game [11] in order to balance the tradeoffs between exploitation of known patterns and exploration of new and changing spectrum dynamics.

Implementation & Results: We have implemented SpecInsight using two USRP radios [8], equipped with the SBX and WBX daughterboards.[1] Our prototype senses over 4 GHz of spectrum, from 50 MHz to 4.4 GHz. We have compared SpecInsight with a setup that uses exactly the same hardware but sequentially scans the spectrum (similar to the Microsoft Spectrum Observatory). Our results show that the probability of missing active signals is 10× lower with SpecInsight when compared to sequential scanning.

We have used the prototype to sense the spectrum in seven locations, including three major US cities and four suburban areas. We report the results of analyzing one week of data from each location and comparing their spectrum patterns. Our main findings are:

- Large swaths of the spectrum may appear completely empty when they actually have active signals. In particular, about 39% of the bandwidth below 4.4 GHz is used by signals whose occupancy is less than 0.0001, and hence are typically invisible to sequential scanning.
- One may think that the common way the spectrum is used is highly dynamic – i.e., a source may transmit at any time. We found that about 65% of the spectrum utilization patterns are either always on, or transmit periodically. Further, among the dynamic patterns, only 5% are highly dynamic[2]. Thus, knowing the spec-

[1] The use of two radios is not fundamental to our design but rather imposed by the range of frequencies of the USRP daughterboards.

[2] Defined as having a standard deviation of when the signal will next appear that exceeds 200ms.

trum patterns is highly useful for smart scheduling of sensing activities.

Contributions

- SpecInsight is, to our knowledge, the first spectrum sensing system capable of detecting and tracking fleeting signals (whose occupancy is ~10^{-5}) in multi-GHz spectrum, while using only tens of MHz of instantaneous bandwidth. Past systems have not been able to combine specificity with scalability: they either provide detailed spectrum occupancy in a single band, e.g., ISM [25, 14], or they obtain coarse occupancy data but miss low-occupancy signals like those in Figures 1b and 1c [21].
- SpecInsight introduces an innovative algorithm for learning spectrum usage patterns, and a smart scheduling algorithm for tracking the presence, absence, and variations of these patterns in realtime over a wide bandwidth of 4 GHz.
- The paper presents a large scale study of spectrum usage patterns in 7 US locations that span urban and suburban areas, illustrating which signal patterns appear in which parts of the spectrum.

2 Related Work

Past work on spectrum sensing may be divided into narrow-band and wide-band techniques. Narrow-band techniques assume the radio bandwidth is at least as wide as the sensed band. They focus on ways to accurately detect a signal. They may use energy level [31], cyclostationarity [14], signal waveform [34], wavelet transform [27], or response to interference [23]. Wide-band sensing techniques try to cover a wide spectrum significantly larger than the radio's own bandwidth. The traditional approach scans the spectrum sequentially and reports average occupancy [21, 34]. Some recent proposals exploit the sparsity of spectrum utilization to sense the spectrum without sampling it at the Nyquist rate, leveraging techniques like compressive sensing [24, 4] or the sparse FFT [12, 10, 13]. For example, BigBand [13] is able to recover the full signals in the spectrum, but under a sparsity assumption that only a small fraction of the spectrum is occupied, so it cannot be used in crowded spectrums, e.g., under 1.5GHz. Another scheme, QuickSense [33], employs a hierarchical search algorithm and analog filters to sense the white spaces, which spans only hundreds of MHz where the wireless technologies are mostly documented.

SpecInsight is a wide-band spectrum sensing technology. SpecInsight, however, differs from the above work in that it does not need sparsity assumptions or custom analog filters. Additionally, SpecInsight covers a wider band than this prior work and provides details of the usage patterns in each band (frequency hopping, periodic, continuous in time but not in frequency, etc.).

SpecInsight also builds on past work that proposed the use of sensing history for dynamic spectrum access [34]. Specifically, a series of theory papers [17, 36] models the behavior of primary users as a Markov process [36] and predicts future opportunities for dynamic spectrum access. SpecInsight differs from these past proposals both in objective and technique. Specifically, while they focus on finding some portion of the spectrum that is idle, SpecInsight focuses on exhaustively characterizing all active signals in the entire spectrum. As a result, the algorithms SpecInsight uses for characterizing historical patterns and scheduling sensing operations differ from the models in past work. Also, SpecInsight is focused on practical system design and empirical data and is supported by a spectrum study that spans multiple locations in the US.

Another line of work focuses on collaborative sensing, where different nodes share spectrum data in order to cover a large geographical area. For example, SpecNet [16] uses spectrum analyzers in different locations to sense the spectrum and share their results; V-Scope [35] mounts spectrum sensors on public vehicles and leverages mobility to enable large-area sensing of the white spaces. SpecInsight complements these systems by enabling *multi-GHz* spectrum sensing on relatively low-cost and easily accessible USRP radios.

Our work is also related to past literature on signal feature extraction. Many of these systems are focused on the ISM band with the objective of identifying WiFi interferers [19, 25, 14]. SpecInsight builds on the idea of signal feature extraction. However, it differs both in the features it extracts and the algorithm it uses to extract them. These differences stem from SpecInsight's use of features to identify spectrum utilization patterns that can be leveraged for smart scheduling of sensing operations, rather than to identify particular technologies. Additionally, SpecInsight spans a 40× wider band than the ISM band, and hence has to deal with a greater diversity of wireless techniques, of which the majority are undocumented.

Finally, our work supplements past work on large-scale spectrum measurements [6, 18, 26, 15]. First, our findings about spectrum occupancy and usage confirm many past spectrum observations; Second, by enabling wide-band spectrum sensing on low-cost devices, we believe SpecInsight opens up the possibility of even larger scale spectrum measurements.

3 SpecInsight's Design

The goal in designing SpecInsight is to build a tool for sensing spectrum usage, extracting occupancy patterns,

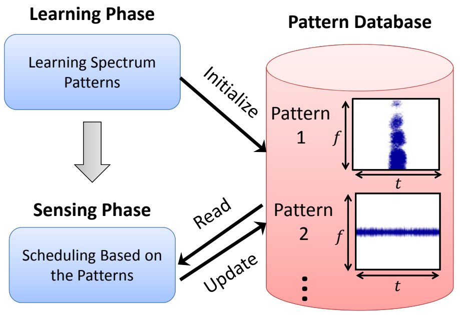


Figure 2: Flowchart of SpecInsight's Architecture: SpecInsight has two phases: the learning phase and the sensing phase. In the learning phase, SpecInsight extracts and learns the patterns in the spectrum and initializes the pattern database; in the sensing phase, SpecInsight uses the learned patterns to schedule when to sense each band. The pattern database stores and maintains the learned patterns, which are representative frequency-time blocks of the underlying signal.

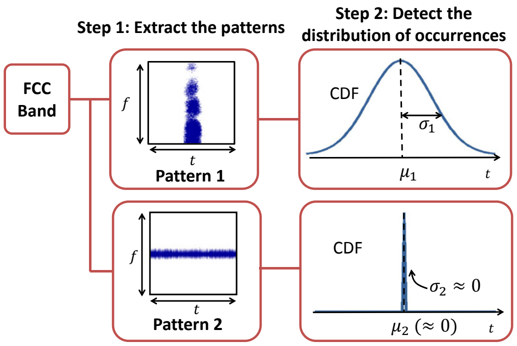


Figure 3: SpecInsight's Learning Phase: To extract pattern information in any given FCC band, SpecInsight employs two steps in the learning phase: 1) extract the patterns; 2) detect the distribution of occurrences of the patterns. The patterns extracted by SpecInsight, as well as the distributions of their occurrences are stored in the pattern database.

and detecting their repeated occurrences. Its key feature is the ability to provide realtime occupancy information of 4 GHz of spectrum using inexpensive commodity radios whose realtime bandwidth is limited to tens of MHz (e.g., USRPs). Anyone can download the SpecInsight software, deploy it on a USRP radio, and start sensing GHz of spectrum in their location.It not only senses a large bandwidth, but also provides finer details at each frequency, so that domain experts in each band can look into the spectrum patterns captured by SpecInsight for further analysis. We envision that such a system will help make wide-band spectrum sensing ubiquitous.

SpecInsight operates in two phases: a learning phase and a sensing phase. During the learning phase, SpecInsight sequentially scans the entire spectrum. It uses the collected data to extract and learn the different usage patterns which it then stores in a pattern database as shown in Fig. 2. Once the database has been populated with the usage patterns of each frequency band, SpecInsight goes into the sensing phase. It uses a smart scheduling algorithm to pick the best frequency band to sense based on the learned patterns. SpecInsight then collects signals in the chosen band and uses a pattern recognition algorithm to decide if the signals belong to a known usage pattern. If not, SpecInsight continues sensing that frequency band for an extended period to learn new usage patterns and update the pattern database.

What are the *patterns*? Spectrum *patterns* are a key concept in SpecInsight's design. A pattern is a representative time-frequency block which characterizes the underlying signal in both time and frequency dimensions. In the example of Fig. 2, pattern 1 spans the whole frequency bandwidth but is narrow in time, while pattern 2 reveals a utilization that is continuous in time but occupies a narrow bandwidth in frequency. The question now is, how do we determine the frequency and time widths of these blocks? On the frequency axis, SpecInsight sets the frequency range of a given pattern equal to one block in the FCC spectrum allocation table [2]. On the time axis, SpecInsight is presented with a trade-off: a short duration allows us to better detect fleeting signals while a long duration allows us to capture longer signals that repeat at a much larger time granularity. To be able to capture both types of signals, SpecInsight uses both short and long time durations. Specifically, in our implementation, we use durations of 5 ms and 50 μs.

For each time-frequency block as defined above, SpecInsight normalizes its power so that the maximum power is equal to 1. This is necessary since two wireless users with the same usage pattern can have significantly different power levels due to different signal attenuations from these users to SpecInsight's sensing antenna. Thus, if we do not normalize, two time-frequency blocks with the same usage pattern can be misidentified as two different patterns. Normalizing also allows us to match time-frequency blocks measured at different spatial locations which allows us to discover similar usage patterns across different urban and suburban areas.

Next, we describe how SpecInsight learns these patterns and uses them to schedule its sensing of each band.

4 The Learning Phase

In the learning phase, SpecInsight extracts and learns information of the spectrum patterns. This process is summarized by Fig. 3. Since SpecInsight divides the frequency spectrum into FCC bands according to the FCC allocation table, we focus only on a single FCC band in the following discussions. First, SpecInsight extracts patterns that exist in this band. Because some FCC bands (e.g., the ISM band) are shared by different types of signals, there might be more than one signal pattern in the band. In this case, SpecInsight extracts and records all of the patterns it can capture. Second, as shown in Fig. 3, SpecInsight keeps track of when each pattern repeats it-

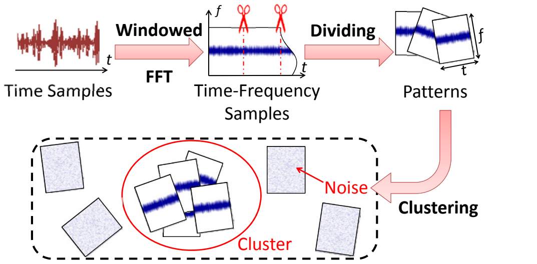


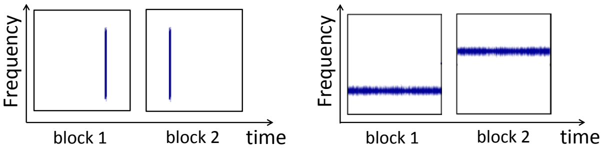
Figure 4: How SpecInsight Extracts Patterns in One FCC Band

self and draws the distribution of the time intervals between different occurrences of the same pattern. This distribution characterizes the timing properties of the underlying signal, e.g., a fixed-cycle signal would have a concentrated distribution while a dynamic signal would have a scattered distribution. SpecInsight stores the list of existing patterns and its corresponding distribution of occurrences in the spectrum pattern database. In the following two subsections §4.1 and §4.2, we describe these two steps in detail.

4.1 Extracting the Patterns

Fig. 4 outlines how SpecInsight extracts the patterns and identifies patterns from noises in a given FCC band. Since patterns are in the form of time-frequency blocks of signals, SpecInsight needs to first transform the I/Q time samples output [3] by the sensing hardware into two-dimensional time-frequency samples. SpecInsight does this by taking the FFTs over a sequence of successive time windows to obtain *time-frequency blocks*.[4] However, not all time-frequency blocks extracted by SpecInsight represent actual signals. Some of them might just be noise. So, how can SpecInsight tell signal patterns apart from noise? The intuition is that wireless signals intrinsically have certain regularities in the way that they use the spectrum, which are reflected by the time-frequency blocks SpecInsight extracts. On the other hand, noise is random. So if we run a clustering algorithm on the time-frequency blocks collected by SpecInsight, signal patterns will be clustered and noise will be filtered out. [5]

There may be multiple spectrum patterns in the same FCC band. In such scenarios, the clustering algorithm can also distinguish between the different patterns, i.e., blocks belonging to each utilization pattern are clustered together and separated from others. This is essential for SpecInsight's sensing phase, because the smart scheduling algorithm has different scheduling strategies for signals with different patterns (e.g., fixed-cycle or dynamic in time). Often each pattern ties to a specific technology, e.g., WiFi and Bluetooth are clustered to two different patterns; however, the goal of distinguishing different patterns is not to precisely identify wireless technologies, but to separate different patterns of spectrum utilization to sense the spectrum more efficiently.



(a) Two time-frequency blocks of a fleeting signal

(b) Two time-frequency blocks of a frequency-hopping signal

Figure 5: Examples where Euclidean distance fails

4.1.1 *Clustering Metric*

Our clustering algorithm needs a distance metric in order to group time-frequency blocks into different clusters of usage patterns, where a small distance between two blocks means they are likely to be in the same cluster. A straightforward solution would be to use the Euclidean distance between two blocks. However, Euclidean distance does not work for some signals (e.g., the ones showed in Fig. 5 (a) and (b)), because it does not take into account possible shifts in the signals. For fleeting signals, the time pulse can appear at any time shift within each time-frequency block; for the frequency hopping signals, the center frequency in each time-frequency block can be different.

To solve this issue, we compute the shifted correlation between two time-frequency blocks. We shift the time-frequency blocks in both time and frequency and pick the minimum Euclidean distance across all shifts as our clustering metric. Formally, given two time-frequency blocks $B_1(f,t)$ and $B_2(f,t)$, our clustering metric is:

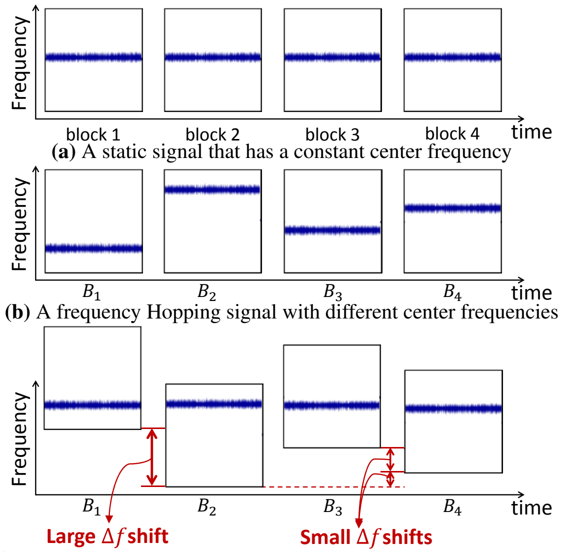
$$D(B_1,B_2) = \min_{\Delta f,\Delta t} \sum_{f,t} |B_1(f,t) - B_2(f-\Delta f, t-\Delta t)|^2 \quad (1)$$

where Δf and Δt represent any possible shift in frequency and time respectively. Using the above metric, we are now able to correctly cluster together the two time-frequency blocks in Fig. 5(a) and Fig. 5(b). Unfortunately, while the shifted distance metric solves the issue in Fig. 5, it creates a new problem that it can render two different usage patterns indistinguishable. For example, consider the two usage patterns in Fig. 6. Fig. 6(a) shows four time-frequency blocks of a frequency band with a static signal that has the same center frequency all the time and Fig. 6(b) shows four time-frequency blocks of a frequency band with a dynamic signal that hops from one center frequency to another. For any pair of time-frequency blocks in Fig. 6(a) and (b), the above distance metric will be small since the shifted correlation will

[3] I/Q samples are the real and imaginary parts of the time samples.

[4] SpecInsight also squares the magnitude since blocks are represented in terms of their powers.

[5] Some signals like the direct spread-spectrum signals which are below the noise floor will not be captured by SpecInsight. However, without prior knowledge of the spreading codes, any energy-based detection will likely miss these signals.



(a) A static signal that has a constant center frequency

(b) A frequency Hopping signal with different center frequencies

(c) Aligning the time-frequency blocks of a frequency hopping signal

Figure 6: The Shifted Euclidean Distance

align the center frequencies in the blocks with the hopping signal (demonstrated in Fig. 6(c)). Hence, all these time-frequency blocks will be clustered together as the same pattern, while they are actually different signals.

To solve this problem, we constrain the time and frequency shift of the time-frequency block to a small range. Instead of computing the Euclidean distance in Eq. 1 for all values of Δt and Δf, we compute it only for a small range of Δt and Δf. To see how this approach can solve this problem, consider again the four blocks (B_1,B_2,B_3, and B_4) which contain a frequency hopping signal shown in Fig. 6(b). By constraining the shift, the distance metric between blocks B_1 and B_2 now becomes large because the center frequencies in B_1 and B_2 are far apart and cannot be aligned with a small shift as can be seen from Fig. 6. However, the distance metric between blocks B_1 and B_3 remains small since the center frequencies are near and can be aligned with a small shift. Thus, for a frequency hopping pattern, some pairs of blocks will have a small distance metric and some pairs will have a large distance metric. This will allow us to distinguish this usage pattern from the static usage pattern shown in Fig. 6(a) where all pairs of blocks have the same small distance metric.

The main question, however, becomes: *If two time-frequency blocks like B_1 and B_2 in Fig. 6(b) have a large distance metric, how can we still cluster them together?* Although B_1 and B_2 have a large distance metric, they are linked together via a chain of blocks that have small distance metrics. In other words, B_1 has a small distance metric with B_3 which in turn has a small distance metric with B_4 which has a small metric with B_2. Thus, although some of these blocks have large distance metric, they are still linked together via a *chain structure* which allows us to cluster them correctly as we will discuss in §4.1.2.

4.1.2 Clustering Algorithm

Machine learning provides us with a rich body of clustering algorithms. However, many of the well-known clustering algorithms such as the k-means do not work for this application. These algorithms are going to cluster together time-frequency blocks that have a small distance metric. As a result, they are not capable of capturing the *chain structure* cluster described above, where two blocks can have a large distance metric and yet belong to the same cluster. Thus, we need a clustering algorithm that is capable of clustering these chain structures.

To this end, we use the OPTICS algorithm [5]. This algorithm achieves exactly the above goal. At a high level, OPTICS is built on the concept of "reachability". Two time-frequency blocks are directly linked together if they have a small distance metric. Two other blocks B_1 and B_2 belong to the same cluster if there is a path of blocks that links B_1 to B_2. For example, in Fig. 6(b), the path was B_1,B_3,B_4,B_2. Thus, a cluster can be interpreted as a set of time-frequency blocks such that any pair of blocks can reach each other. Another advantage of the OPTICS algorithm over the k-means is that it does not require the number of clusters as an input. For the exact details of the OPTICS algorithm, we refer the reader to [5].

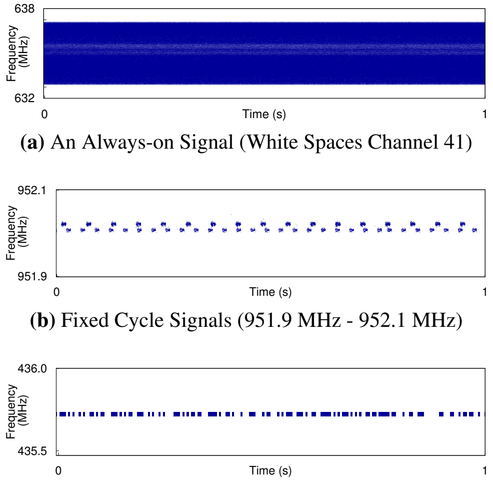
SpecInsight uses the OPTICS algorithm in two places:

- During the learning phase: SpecInsight runs the full OPTICS algorithm to cluster the collected usage patterns and establish a pool of patterns. The number and types of classes is data dependent. In §8, we describe the classes of usage patterns which are revealed by our experiments.
- During the sensing phase: SpecInsight uses OPTICS to cluster the newly sensed usage pattern and determine whether they belong to an already learned cluster of usage patterns or they form a new cluster of patterns that needs to be added to the pattern database.

4.2 Detecting the Distribution of Occurrences

Once SpecInsight extracts and identifies a specific pattern, it tracks the different times when the pattern recurs and builds an occurrence distribution (step 2 in Fig. 3). SpecInsight defines the *pattern interval* τ as the time between two consecutive occurrences of the pattern, and the distribution of occurrences is defined as the statistical distribution of the pattern interval τ. It can be characterized by its mean μ and standard deviation σ, which SpecInsight computes over multiple measurements.

These statistics μ and σ are necessary to sense the spectrum efficiently. The mean μ determines the period of the pattern, and the standard deviation σ measures how dynamic the signal is. Thus, μ can be used to decide



(a) An Always-on Signal (White Spaces Channel 41)

(b) Fixed Cycle Signals (951.9 MHz - 952.1 MHz)

(c) A Dynamic Signal (940 MHz - 940.5 MHz)

Figure 7: Examples of Usage Patterns Over Time: Three types of signals are shown according to their timing characteristics: always-on signals, fixed-cycle signals and dynamic signals.

how often and at what time we expect to see the signal and σ tells us how precise our prediction is and can be used to decide the duration over which we should sense the band.

These distributions of pattern occurrences, as well as the pool of patterns that SpecInsight extracts and identifies, are stored in SpecInsight's spectrum pattern database (Recall Fig. 3 for an outline of what is in the database). In the following section, we will expand on how SpecInsight's sensing phase can utilize this database to sense the spectrum efficiently.

5 The Sensing Phase

After the pattern database is initialized in the learning phase, SpecInsight goes into the sensing phase and uses a smart scheduling algorithm to decide which frequency band to sense at each given time. Before we delve into the details, we will start with an example that gives some intuition behind SpecInsight's smart scheduling algorithm.

5.1 Intuition

SpecInsight's scheduling strategy builds on the following key intuitions. First, if a signal appears regularly every period, it will be much easier to catch this signal at its next predicted period even if it is a short fleeting signal. Second, we should spend more time sensing frequency bands with dynamic usage patterns and minimize the time we spend sensing bands with usage patterns that are static or have little uncertainty.

To better understand why this makes sense, let us consider three simple examples of usage patterns that have different time properties (i.e., their distributions of occurrences are very different): 1) always-on signals ($\mu \approx 0, \sigma \approx 0$) as in Fig. 7(a), 2) fixed cycle signals ($\sigma \approx 0$) as in Fig. 7(b), and 3) dynamic signals (σ is large) as in Fig. 7(c). Intuitively, for always-on signals, we can scan the frequency band less often in order to check from time to time that the signal is still there. For fixed-cycle signals, we can predict exactly when the signal is going to appear and sense the band precisely at that time. We also might want to check at times when we predict the band to be idle in case our prediction is wrong and there is another user using the band with a different usage pattern. For dynamic signals, the best strategy would be to sense the band at random times but for longer durations. We can afford to sense these bands for longer time given the time we saved on bands with always-on and fixed-cycle signals.

This gives the intuition. In the following section we will formalize this intuition into the *smart scheduling algorithm* that SpecInsight employs in its sensing phase.

5.2 The Smart Scheduling Algorithm

The smart scheduling algorithm needs to answer two main questions:

- Which frequency band f to sense next?
- How long to stay in a frequency band f?

Which frequency band f to sense next? Answering this question requires balancing a trade-off between exploitation and exploration. On one hand, we can exploit the information we learned from the sensing history to schedule brief checks on the next occurrence of a signal in some frequency band. On the other hand, due to the dynamics of the spectrum, the history information we have might not be accurate. So we need to keep exploring the spectrum in order to discover new usage patterns.

To address this trade-off, we formulate the problem as a *multi-armed bandit game* [11]. The multi-armed bandit game is a well studied problem in decision theory. In this game, the gambler needs to iteratively choose from K bandit machines, each of which will give her random rewards according to an unknown distribution. Her goal is to maximize the rewards in a given number of rounds. The gambler could learn the distribution by repeatedly pulling the levers. She then needs to decide whether to exploit the information she learned and choose the lever that maximizes her expected payoff or to just explore more in order to better learn the distribution.

There is a large literature of solutions to the multi-armed bandit game [30, 11]. In our implementation, we adopt a simple but very effective solution called *the ε-greedy strategy* which provides a very good approximation to the optimal decision [30]. In this solution, gambler simply chooses the lever that maximizes her expected payoff for $(1-\varepsilon)$ of the time and for the remaining ε of the time she picks a lever at random. The choice

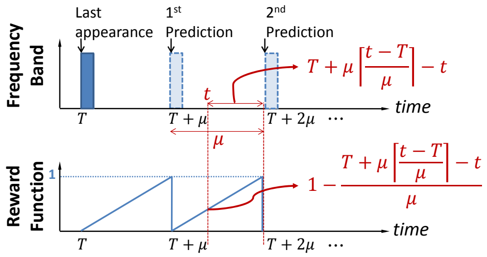

Last appearance
1st Prediction
2nd Prediction
Frequency Band
$T + \mu \left\lceil \frac{t-T}{\mu} \right\rceil - t$
t
time
T
T + μ
T + 2μ ···
μ
Reward Function
1
$1 - \frac{T + \mu \left\lceil \frac{t-T}{\mu} \right\rceil - t}{\mu}$
time
T
T + μ
T + 2μ ···

Figure 8: The Reward Function: The reward function $R_f(t)$ shows how near we are to the next signal appearance. It is zero at the beginning of a predicted period and one at the end, while linearly increasing as we get nearer to time we predict the signal will appear.

of ε defines the degree to which we rely on the learned information and ε is traditionally set to 0.1 [30].

Thus, 10% of the time, SpecInsight is going to pick a random frequency band to sense and 90% of the time, it will pick the band that gives it the maximal reward. But what is the reward function that SpecInsight needs to maximize? SpecInsight avoids missing a signal by going to its frequency band just before it expects the signal to appear. As a result, SpecInsight uses an indication of how near we are to the next expected appearance of a signal in the frequency band as its reward function. Formally, we calculate the reward function for a frequency band f at time t as:

$$R_f(t) = 1 - \frac{T + \mu\lceil (t-T)/\mu \rceil - t}{\mu} \qquad (2)$$

where T is the last time the signal was observed and μ is the mean value of the pattern interval time as described in §4.2. The reward function is normalized to 1 in order to compare bands with different mean pattern interval μ.

To better understand this reward function, consider the example shown in Fig. 8. Given the last appearance of a signal at time T and the expected cycle μ, we predict the signal will appear again at times $T+\mu, T+2\mu, T+3\mu, \cdots$. Thus at time t, we predict that the signal will appear next at time $T+\mu\lceil (t-T)/\mu \rceil$ and we are $T+\mu\lceil (t-T)/\mu \rceil - t$ away from it. Since the farthest we can be away from the next appearance is μ, we normalize by μ and subtract it from 1 so that the nearer we are, the larger the reward function is.

How long to stay in a frequency band f? Once SpecInsight decides which frequency band to sense, it needs to decide how long to stay in that band. We refer to this as the dwell time t_d. The dwell time is determined by the number of measurements (time-frequency blocks) we need to collect in each band. It is directly related to the dynamics of the pattern, for the following reason: The more dynamic the usage pattern is, the more uncertain we are of our predictions, so that the offset between the predicted occurrence of the signal and the actual occurrence is bigger. To compensate for that, we need to have longer measurement time in order to capture the signal. As a result, the number of measurements needs to be proportional to the uncertainty in our predictions of when the signals are going to appear.

Algorithm 1: Smart Scheduling Algorithm

```
Procedure SMARTSCHEDULING($\{f\}$, $\{\mu\}$, $\{\sigma\}$, $\{T\}$)
  $t \leftarrow$ Current Time
  if RAND($[0,1]$) $< \varepsilon$ then                ▷ The $\varepsilon$-greedy strategy
    $f^* \leftarrow$ RAND($\{f\}$)                    ▷ Pick random frequency
  else
    for $f$ in $\{f\}$ do
      $\mu, T \leftarrow \{\mu\}_f, \{T\}_f$
      if $\mu \neq 0, \infty$ then
        $R_f(t) \leftarrow 1 - \frac{T+\mu\lceil (t-T)/\mu \rceil - t}{\mu}$
      else
        $R_f(t) \leftarrow$ RAND($[0,1]$)
    $f^* \leftarrow \arg\max_f R_f(t)$
  $t_d \leftarrow \min\{6\{\sigma\}_{f^*}$, small constant$\}$
  return $\{f^*, t_d\}$
```

The dynamics of the pattern, i.e., the level of uncertainty, is captured by the standard deviation σ of the pattern interval τ which SpecInsight extracts in the learning phase. The bigger σ is, the more dynamic the usage pattern is. SpecInsight uses the *3-Sigma Rule* [28] to determine the dwell time t_d. The rule states that a $\pm 3\sigma$ interval centered at the mean of the distribution covers most of the cases. For example, in a Gaussian distribution, it covers 99.7% of the probabilities. More generally, for any distribution it covers at least 90%. Based on this rule, SpecInsight sets the dwell time to be $t_d = 6\sigma$.

A few points are worth noting:

- The reward function in Eq. 2 is not well defined for frequency bands with always-on usage patterns where $\mu = 0$ and for frequency bands with no signals where $\mu = \infty$ (always idle). For these frequency bands, we pick the reward function randomly between 0 and 1.
- Frequency bands with fixed-cycle signals, always-on signals, or no signals have $\sigma \approx 0$. For these bands, we set a minimum dwell time t_d such that the collected data contains at least a few time-frequency blocks.
- Some frequency bands might contain multiple patterns, where each pattern has its own μ and σ. SpecInsight randomly picks one of the usage patterns' μ and σ to calculate the reward function and the dwell time.
- In the case of fixed-cycle signals, SpecInsight is able to track the signals while sequential scanning only detects the signal with some probability. Our ability to track the signals is important in the case of fleeting periodic signals like the one in Fig. 1(c), which are very easy to miss using sequential scanning.
- Finally, SpecInsight is a best-effort system and might miss sensing deadlines if pattern dynamism in the entire spectrum is very high. In the worst case, if all

of the bands in the spectrum were equally highly dynamic, it would degrade to randomly sampling the bands but would still be no worse than sequential scanning. Fortunately, as we will show in section §8, only very few ($< 5\%$) of the patterns are highly dynamic in today's spectrum and SpecInsight works well.

Finally, a pseudocode of SpecInsight's smart scheduling algorithm is shown in Alg. 1.

6 Implementation

We implement SpecInsight on USRP software radios [8]. Since each USRP daughterboard works in a particular frequency range, we use two USRPs that simultaneously run SpecInsight: the first USRP is equipped with an SBX daughterboard, and works in the frequency range from 400 MHz to 4.4 GHz, and the second USRP is equipped with a WBX daughter-board and works in the frequency from 50 MHz to 2.2 GHz. We connect the two USRPs to the same antenna using a power splitter. We use an ultra-wideband omni-directional outdoor antenna that works from 25 MHz to 6 GHz [22].

In order to maximize the USRP capabilities, we tune the bandwidth and sampling rate to their maximum (40 MHz and 50 Ms/s). We set the two USRPs to sense non-overlapping frequency ranges, i.e, 50 MHz to 2.2 GHz and 2.2 GHz to 4.4 GHz. Each of them runs an independent version of SpecInsight's sensing algorithm, and their spectrum pattern databases are combined together. Thus, SpecInsight senses a total spectrum bandwidth of 4.35 GHz, from 50 MHz to 4.4 GHz. SpecInsight divides this spectrum into 171 bands based on the FCC spectrum allocation table [2]. For each band, it learns its spectrum patterns and schedules when to sense the band according to the algorithms in §4 and §5.

Implementing SpecInsight in realtime is challenging. SpecInsight needs to process a data stream over a Gbit/s. In order to support such high data rates, we implement all major computations using Intel's streaming SIMD extension (SSE2) instruction set, which provides instruction-level parallelization. We also use the FFTW library [3] for fast FFT implementation. Consequently, we are able to run SpecInsight in realtime on a machine with an 8-core Intel-i7 processor and 8 GB of RAM.

7 USRP Calibration

SpecInsight is not hardware specific, and can be used with various radios. The radio hardware, however, may have its own spurs, i.e., fake signals generated by hardware noise, which might be recognized by SpecInsight as patterns. Thus, when running SpecInsight on a particular hardware platform, the radio should be calibrated to identify hardware-specific spurs and eliminate them.

We calibrate the USRPs used in our prototype. All of our calibration experiments are conducted in a Faraday shield room which blocks all signals from the outside.

Calibration in the absence of signals: We put our sensing setup in the shield room, and collect measurements in the absence of any transmission. Since all active signals from the outside are blocked by the room, every received signal that is above the noise floor is a spur from the hardware. We noted two types of USRP spurs: 1) the USRP always shows power at the baseband DC frequency, 2) the time samples received during the first 10ms after power-on are corrupted. We add filters to SpecInsight to remove these spurs before running the algorithms. After adding these filters, SpecInsight does not detect any pattern in the samples collected by the USRPs in the shield room. This complies with the fact that there are no active signals in the environment, and random noise is discarded by the pattern clustering algorithm.

Calibration in the presence of transmission: USRPs do not adapt the receiver's gain with the signal power. As a result, signals whose power is higher than the ADC's maximum quantization level are clipped at the receiver. Clipping distorts the received signal and changes its frequency representation (creating harmonics). To ensure that the received signal's frequency representation matches that of the signal over the air, the receiver should be operating in its linear range without clipping.

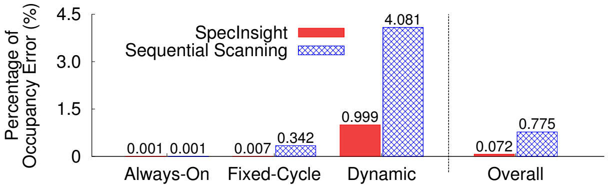
The common approach to avoid clipping is to add automatic gain control (AGC) to the receive chain [7]. USRPs however do not implement AGC. To address this issue, SpecInsight detects the occurrences of clipping by counting the number of time samples that are equal to the maximum quantization value. Once clipping is detected, SpecInsight drops the samples and sends out alerts. During our experiments, which encompass 7 locations and a total of 49 days, we noted only 7 occurrences of clipping, which were removed from the data. Please note that the clipping problem is specific to our sensing hardware but not fundamental to the algorithm; to avoid it, one could use a more expensive hardware that implements AGC.

We run experiments in the shield room with a transmitter to check SpecInsight's ability to detect a pattern correctly and eliminate clipping events. We let the transmitter transmit continuously, but vary its transmission power. We confirm that SpecInsight detects the signal in the correct frequency band as long as there is no clipping, and generates an alert whenever the signal clips.

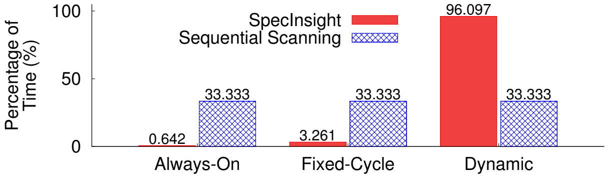
8 Empirical Results

8.1 SpecInsight's Accuracy

We compare SpecInsight with a setup that uses exactly the same USRP hardware but sequentially scans the



(a) Percentage of Occupancy Error



(b) Percentage of Time Spent in Each Class

Figure 9: Comparison of SpecInsight with Sequential Scanning: (a) shows that overall SpecInsight reduces errors by 10× in comparison to sequential scanning; (b) shows that SpecInsight uses its time wisely spending less time on always-on and fixed-cycle bands and more time on dynamic bands.

spectrum, as typical in today's systems [21]. For sequential scanning, the dwell time of each band is set to 50ms, which matches the average dwell time of SpecInsight.

To compare the accuracy of the two systems, we need the ground truth. However, existing sensing hardware does not have 4 GHz of instantaneous bandwidth thus cannot provide the ground truth for such a wideband. To address this issue, we use 10 USRPs to continuously monitor a subset of the bands within the 4 GHz spectrum, and obtain their ground truth. This provides us with the ground truth needed to calculate the accuracy of SpecInsight and sequential scanning for this particular sub-set of bands. We then repeat the experiment for different subsets of bands.

We categorize the bands based on their usage patterns to: always-on (on for $> 95\%$ of the time), fixed-cycle ($\sigma < 5$ms), and dynamic ($\sigma > 100$ms). In our experiments, we consider equal number of bands (20) of each type; for each band we run the experiment for 1 hour. For both SpecInsight and sequential scanning, we compute the following two metrics for each type of bands:

- **Percentage Occupancy Error:** This is the percentage difference between the ground truth occupancy of a band and the occupancy reported by SpecInsight and sequential scanning. We define occupancy as the percentage of time the band is occupied.
- **Percentage of Sensing Time:** This is the percentage of the total amount of time that the sensing algorithm spends in each type of band.

Results: The results using the above two metrics are shown in Fig. 9. For always-on bands, SpecInsight spends 50× less time in these bands and still achieves the same accuracy as sequential scanning. For fixed-cycle bands, SpecInsight spends 10× less time in these bands and yet has 50× higher accuracy. For bands with more dynamics, SpecInsight can afford to spend 2.5× more time in these bands which translates into 4× higher accuracy. Finally, overall, *SpecInsight has 10× higher accuracy than sequential scanning for the same time budget.* This is due to its smart scheduling algorithm, which spends as little time as needed on always-on and fixed-cycle signals, and saves its time for dynamic signals.

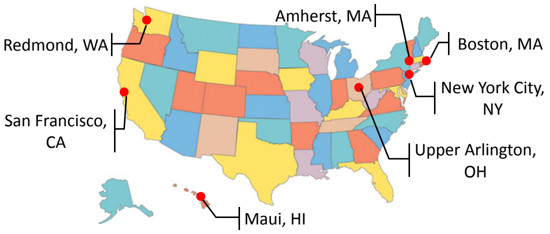


Figure 10: SpecInsight's Measurement Locations.

8.2 Real-World Spectrum Analytics

We deployed SpecInsight in seven locations in the US, including three major cities and four suburban areas, which cover the East Cost, West Cost and Pacific islands (Fig. 10). In each location, we analyzed one week of data collected by SpecInsight. We report the results below.

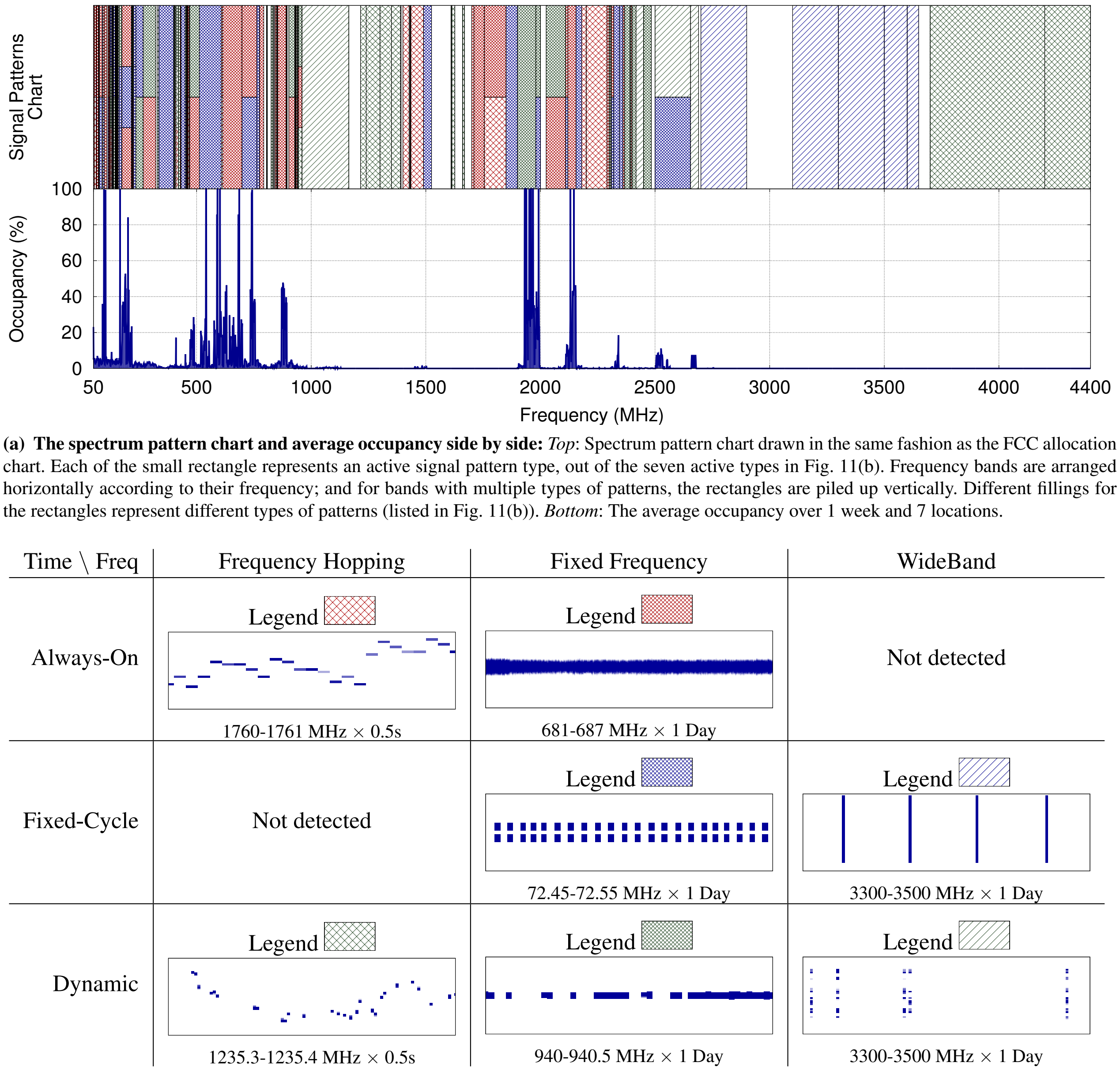
8.2.1 The Spectrum Pattern Chart

In this section, we want to analyze how the spectrum usage patterns are distributed across frequencies. Over one week and seven locations, SpecInsight detected a total of 312 different patterns corresponding to different technologies. To be able to visualize these patterns, we group them into classes according to their time and frequency properties. In the time dimension, we divide the patterns into always-on, fixed-cycle and dynamic. In the frequency dimension, we divided the patterns into frequency-hopping, fixed frequency, and wideband [6]. This gives us a total of 3×3=9 classes [7], where Fig. 11 (b) shows one usage pattern example for each class. Based on these usage patterns, we constructed the first-of-its-kind *spectrum pattern chart* shown in Fig. 11 (a). In a similar fashion to the FCC's spectrum allocation chart, the spectrum pattern chart shows the types of spectrum usage patterns seen in different frequency bands. Please note that we group the patterns into these rough classes just for the purpose of visualization; SpecInsight's database contains the exact and detailed patterns in each FCC band, in the form of time-frequency blocks.

Results: Fig. 11(a) shows the spectrum pattern chart (top) and the average spectrum occupancy chart (bottom)

[6] We label signals with bandwidth larger than 50MHz as wideband.

[7] Note in all of the experiments we did not see wideband signals that are always on, or frequency hopping signals that repeat in a fixed cycle. Hence, we ended up with a total of 7 classes.



(a) The spectrum pattern chart and average occupancy side by side: *Top*: Spectrum pattern chart drawn in the same fashion as the FCC allocation chart. Each of the small rectangle represents an active signal pattern type, out of the seven active types in Fig. 11(b). Frequency bands are arranged horizontally according to their frequency; and for bands with multiple types of patterns, the rectangles are piled up vertically. Different fillings for the rectangles represent different types of patterns (listed in Fig. 11(b)). *Bottom*: The average occupancy over 1 week and 7 locations.

(b) Legend for rectangle fillings: We divide the patterns according to their time and frequency properties, which are the rows and columns of this chart and each intersection defines a class of patterns. So there are a total of $3 \times 3 = 9$ types. We give examples for 7 types of signals, while the other two (wideband always-on and frequency-hopping fixed-cycle signals) were not detected in any of the 7 locations.

Figure 11: The Spectrum Pattern Chart

over one week and seven locations. The bottom graph is computed by averaging occupancy across locations and the top graph is a superposition of the patterns across all locations. The figure shows that although there are many bands in the occupancy chart that are empty or nearly empty, the pattern chart reveals that these bands are actually being used. For example, the occupancy in the frequency ranges 1.2 GHz–1.85 GHz and 2.9 GHz–4.4 GHz is less than 0.0001 (almost zero). However, SpecInsight detected in these bands some frequency hopping signals and some wide-band fleeting periodic signals. In fact, the figure shows that *although large swaths of the spectrum may appear completely empty, they actually have active signals*. In particular, about 39% of the bandwidth below 4.4 GHz is used by signals whose occupancy is less than 0.0001. Moreover, the usage patterns in these band are mostly of two types: 62.6% are frequency hopping signals and 33.5% are wideband fleeting signals.

To better understand how much bandwidth each type of pattern spans and how much it contributes to the spectrum occupancy, consider Fig. 12. The figure shows the distribution of bandwidths and occupancies of the patterns in government-owned bands, non-government bands and shared bands (where both government and

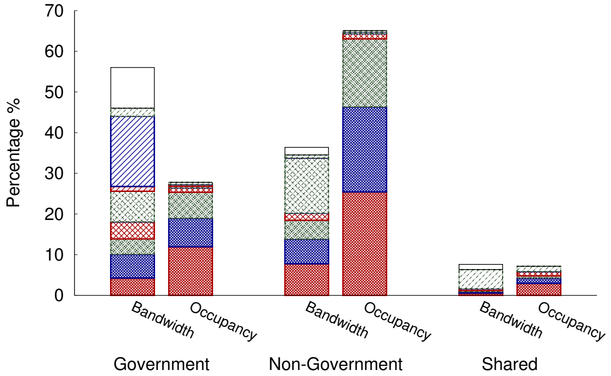


Figure 12: Spanned bandwidth vs. contributed occupancy: We check the spanned bandwidth vs. contributed occupancy for different types of patterns in government-owned, non-government and shared bands. For the legend of this figure, see Fig. 11(b).

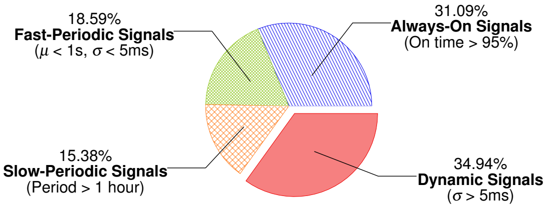


Figure 13: Statistics of Patterns According to the Timing Characteristics: The figure shows that more than half of the patterns (65%) have some timing regularities, either always-on or periodic.

non-government usage coexist). The results reveal that usage patterns like frequency hopping and wideband signals occupy 53.3% of the bandwidth but only contribute 6.8% to the total spectrum occupancy. This is more apparent in government-owned bands since these technologies are typically used in security applications. Particularly, the government owns 56% of the spectrum but only contributes 27.8% to the total occupancy.

8.2.2 *Timing Analytics*

In our timing analysis, we aim to answer the following questions: How many of the spectrum patterns are dynamic? How many are highly predictable (periodic or always-on) signals? We use the standard deviation σ of the pattern intervals (described in §4.2) to distinguish dynamic patterns from periodic and always-on signals. Often higher σ reveals a more dynamic usage pattern. However, this is not always true. Some periodic patterns have a very large period (hours-days), and hence can have a large standard deviation σ. Fig. 14 shows a usage pattern in the government-owned 152 MHz band that repeats every day. In particular, it has a signal that is always present, but at night, it is turned off in every other channel. To accommodate such periodic patterns with large σ, we distinguish between fast periodic and slow periodic.

Results: Fig. 13 shows the percentage of patterns that are always-on, fast-periodic, slow-periodic and dynamic, out

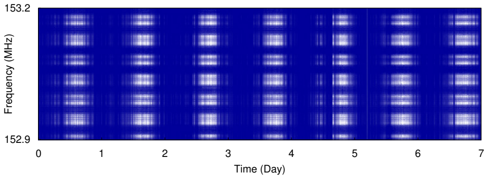


Figure 14: Example of a Slow Periodic Signal: Every other channel of the signal is turned off at night for a fixed duration.

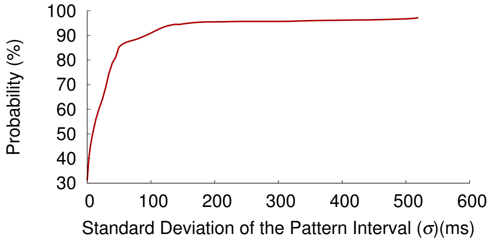


Figure 15: The CDF of the standard deviation of the pattern interval (σ). Only less than 5% of the signals have very large σ.

of the 312 detected patterns. It reveals that *only 35% of the detected usage patterns are actually dynamic*.

To gain more insight into how dynamic the frequency bands are, we compute the CDF of the standard deviation σ of signal intervals. Fig. 15 shows this CDF and reveals that *less than 5% of the patterns are highly dynamic*, i.e., having a very large σ ($\sigma > 200$ms). These results show that knowing the spectrum patterns is highly useful for smart scheduling of sensing activities, and hence the benefits of SpecInsight.

9 Conclusion

This paper presents SpecInsight, a system that can acquire the detailed utilization patterns over 4 GHz of spectrum in real time. We implement SpecInsight using off-the-shelf USRP radios and perform a large-scale study of spectrum analytics in 7 US locations including urban and suburban areas. Consequently, we build the first-of-its-kind spectrum pattern database characterizing how the spectrum is utilized. We believe that SpecInsight enables multiple applications such as dynamic spectrum access, finding breaches of spectrum regulations, and understanding undocumented spectrum utilizations.

Acknowledgments: We thank the anonymous reviewers and shepherd Jacobus Van der Merwe for their constructive comments. We are grateful to Haitham Hassanieh, Anoop Gupta, Aakanksha Chowdhery, Omid Abari, Swarun Kumar, Deepak Vasisht, John MacDonald, Fadel Adib, Hariharan Rahul, Zach Kabelac, Ezz Hamed, and Ranveer Chandra for their constructive feedbacks. We also thank the NETMIT group for their support. This research is funded by NSF. We thank members of the MIT Wireless Center: Amazon, Cisco, Google, Intel, Mediatek, Microsoft and Telefonica for their interest and general support.

REFERENCES

[1] Army use of the electromagnetic spectrum. http://www.apd.army.mil/pdffiles/r5_12.pdf.

[2] United states frequency allocations. http://www.ntia.doc.gov/files/ntia/publications/2003-allochrt.pdf.

[3] FFTW. http://www.fftw.org/.

[4] ABARI, O., LIM, F., CHEN, F., AND STOJANOVIC, V. Why analog-to-information converters suffer in high-bandwidth sparse signal applications. *Circuits and Systems I: Regular Papers, IEEE Transactions on* (2013).

[5] ANKERST, M., BREUNIG, M. M., PETER KRIEGEL, H., AND SANDER, J. OPTICS: Ordering points to identify the clustering structure. In *SIGMOD* (1999), pp. 49–60.

[6] CHEN, D., YIN, S., ZHANG, Q., LIU, M., AND LI, S. Mining spectrum usage data: a large-scale spectrum measurement study. In *Mobicom* (2009).

[7] DRENTEA, C. *Modern Communications Receiver Design and Technology*. Artech House, 2010.

[8] ETTUS. Inc. USRP. http://ettus.com.

[9] FCC. Second memorandum opinion & order 10-174.

[10] GHAZI, B., HASSANIEH, H., INDYK, P., KATABI, D., PRICE, E., AND SHI, L. Sample-optimal average-case sparse fourier transform in two dimensions. In *Allerton* (2013).

[11] GITTINS, J. Bandit processes and dynamic allocation indices. *Journal of the Royal Statistical Society* (1979).

[12] HASSANIEH, H., INDYK, P., KATABI, D., AND PRICE, E. Nearly optimal sparse fourier transform. In *STOC* (2012).

[13] HASSANIEH, H., SHI, L., ABARI, O., HAMED, E., AND KATABI, D. Ghz-wide sensing and decoding using the sparse fourier transform. In *INFOCOM* (2014).

[14] HONG, S. S., AND KATTI, S. R. DOF: A local wireless information plane. In *ACM SIGCOMM* (2011).

[15] ISLAM ET AL., M. Spectrum survey in singapore: Occupancy measurements and analyses. In *CrownCom* (2008).

[16] IYER, A. P., CHINTALAPUDI, K., NAVDA, V., RAMJEE, R., PADMANABHAN, V. N., AND MURTHY, C. R. SpecNet: spectrum sensing sans frontières. In *NSDI* (2011).

[17] KIM, H., AND SHIN, K. Efficient discovery of spectrum opportunities with mac-layer sensing in cognitive radio networks. *Mobile Computing, IEEE Transactions on* (2008).

[18] KONE, V., YANG, L., YANG, X., ZHAO, B. Y., AND ZHENG, H. On the feasibility of effective opportunistic spectrum access. In *Proceedings of the 10th ACM SIGCOMM Conference on Internet Measurement 2010, Melbourne, Australia - November 1-3, 2010* (2010).

[19] LAKSHMINARAYANAN, K., SAPRA, S., SESHAN, S., AND STEENKISTE, P. RFDump: An architecture for monitoring the wireless ether. In *CoNEXT* (2009).

[20] MCHENRY, M. A. NSF spectrum occupancy measurement project summary, 2005.

[21] MICROSOFT SPECTRUM OBSERVATORY. http://spectrum-observatory.cloudapp.net.

[22] MP. Ultra base station antenna. http://www.mpantenna.com/products/product/08-ant-0861-vhf-uhf-basestation-antenna/.

[23] RAHUL, H., KUSHMAN, N., KATABI, D., SODINI, C., AND EDALAT, F. Learning to share: Narrowband-friendly wideband networks. In *ACM SIGCOMM* (2008).

[24] RASHIDI, M., HAGHIGHI, K., PANAHI, A., AND VIBERG, M. A NLLS based sub-nyquist rate spectrum sensing for wideband cognitive radio. In *IEEE DySpan* (2011).

[25] RAYANCHU, S., PATRO, A., AND BANERJEE, S. Airshark: Detecting non-wifi rf devices using commodity wifi hardware. In *ACM IMC* (2011).

[26] TAHER, T., BACCHUS, R., ZDUNEK, K., AND ROBERSON, D. Long-term spectral occupancy findings in chicago. In *IEEE DySPAN* (2011), pp. 100 – 107.

[27] TIAN, Z., AND GIANNAKIS, G. A wavelet approach to wideband spectrum sensing for cognitive radios. In *Cognitive Radio Oriented Wireless Networks and Communications, 2006. 1st International Conference on* (2006).

[28] TRIOLA, M. F. *Essentials of Statistics*. 2009.

[29] PCAST: REALIZING THE FULL POTENTIAL OF GOVERNMENT HELD SPECTRUM TO SPUR ECONOMIC GROWTH, 2012.

[30] VERMOREL, J., AND MOHRI, M. Multi-armed bandit algorithms and empirical evaluation. In *ECML* (2005).

[31] YANG, L., HOU, W., CAO, L., ZHAO, B. Y., AND ZHENG, H. Supporting demanding wireless applications with frequency-agile radios. In *NSDI* (2010).

[32] YING, X., ZHANG, J., YAN, L., ZHANG, G., CHEN, M., AND CHANDRA, R. Exploring indoor white spaces in metropolises. In *ACM MobiCom* (2013).

[33] YOON, S., LI, L. E., LIEW, S., CHOUDHURY, R. R., TAN, K., AND RHEE, I. Quicksense: Fast and energy-efficient channel sensing for dynamic spectrum access wireless networks. In *IEEE INFOCOM* (2013).

[34] YUCEK, T., AND ARSLAN, H. A survey of spectrum sensing algorithms for cognitive radio applications. *Communications Surveys Tutorials, IEEE 11*, 1 (2009).

[35] ZHANG, T., LENG, N., AND BANERJEE, S. A vehicle-based measurement framework for enhancing whitespace spectrum databases. In *Mobicom* (2014).

[36] ZHAO, Q., TONG, L., SWAMI, A., AND CHEN, Y. Decentralized cognitive mac for opportunistic spectrum access in ad hoc networks: A POMDP framework. *Selected Areas in Communications, IEEE Journal on 25*, 3 (2007).

Atomix: A Framework for Deploying Signal Processing Applications on Wireless Infrastructure

Manu Bansal, Aaron Schulman, and Sachin Katti
Stanford University

Abstract

Multi-processor DSPs have become the platform of choice for wireless infrastructure. This trend presents an opportunity to enable faster and wider scale deployment of signal processing applications at scale. However, achieving the hardware-like performance required by signal processing applications requires interacting with bare metal features on the DSP. This makes it challenging to build modular applications.

We present Atomix, a modular software framework for building applications on wireless infrastructure. We demonstrate that it is feasible to build modular DSP software by building the application entirely out of fixed-timing computations that we call *atoms*. We show that applications built in Atomix achieve hardware-like performance by building an 802.11a receiver that operates at high bandwidth and low latency. We also demonstrate that the modular structure of software built with Atomix makes it easy for programmers to deploy new signal processing applications. We demonstrate this by tailoring the 802.11a receiver to long-distance environments and adding RF localization to it.

1 Introduction

Programmable Digital Signal Processors (DSPs) are increasingly replacing ASICs as the platform of choice for performing the heavyweight signal processing in our wireless infrastructure. The primary driver for the shift toward this programmable infrastructure is the increased rate of wireless standard updates: the 3GPP releases a new LTE standard roughly once every 18 months [3]. DSPs enable such quick upgrade cycles since their software can be updated with the push of a button.DSPs in the infrastructure strike balance between high performance, low power, (only 5-8 Watts to run a 20MHz LTE basestation), and programmability. They do so by combining multiple processing cores for programmability with hardware accelerators for performance and power efficiency.

The trend toward building programmable DSPs into wireless infrastructure presents a valuable opportunity: programmers can build new signal processing applications and quickly deploy them on wireless infrastructure at scale. Such applications encompass both communication and non-communication signal processing. Communication applications include standard wireless protocols (e.g., LTE and WiFi), as well as customized protocols for particular environments (e.g., long distance rural broadband [14, 6]). Non-communication applications include using radio waves for localization [28, 8].

There are three basic primitives that programmers need from the software framework of DSPs to build and deploy signal processing applications: *tapping* into a signal processing chain, *tweaking* a signal processing block, and *inserting* or *deleting* a signal processing block. However, the DSP software framework must present a modular interface to the programmer that supports these primitives. Such modularity is essential to add new applications without needing to modify or understand the full software base which could uptake excessive effort. For example, tweaking a specific block to improve its function or efficiency should not affect other blocks, nor should it require the programmer to tweak or understand other parts of the software.

Designing a modular software framework for DSPs is challenging because of the demanding requirements of the communication applications that it must support. Communication applications are both high throughput and latency sensitive. This combination requires hardware-like performance from the DSP software; it must be highly efficient and have predictable execution timing. For example, to process a typical 20MHz WiFi channel, the DSP must process a sample of the signal every 25ns (assuming Nyquist sampling). Assuming the DSP is running at a 1GHz (this is a typical DSP clock rate), there are only 25 cycles available on average per

DSP core to process each sample. Additionally, the processing has to finish within a short time because of ARQ. For instance, WiFi needs to decode and send an ACK within 16μs of receiving the last sample of a packet.

Building modular DSP software that has hardware-like efficiency and predictability is challenging. The primary reason is, programmers must use several bare metal features to achieve hardware-like performance. DSPs such as the TI 6670 [23] are highly parallel processors with multiple cores and hardware accelerators. There is no rich operating system support to manage those resources with adequate performance. As a result, the programmer must manually parallelize software. DSPs typically lack cache-coherent shared memory. This forces the programmer to explicitly move data across cores and hardware accelerators. Further, DSPs tend to have shallow memory hierarchies with software-addressable SRAM that the programmer has to explicitly manage.

In this paper, we present Atomix, a modular software framework for developing high bandwidth and low latency apps for DSPs. The key idea behind Atomix is the *atom* abstraction, which is defined as a unit of execution that takes a fixed, known amount of time to run every time it is invoked. In Atomix, programmers can express every module in a signal processing application as an atom. Atoms can be an algorithmic blocks such as FFT, or compositions of blocks such as OFDM, or different packet processing modes in protocols such as LTE or WiFi, or even low-level plumbing primitives such as data transfer across cores.

We evaluate the Atomix framework by using it to build a standard 802.11a WiFi receiver called Atomix11a. WiFi is an ideal app to evaluate the performance capability of Atomix because it uses the same bandwidth as LTE with similar processing complexity and spectral efficiency of an OFDMA protocol. Additionally, WiFi's decode latency constraints of tens of microseconds are two orders of magnitude tighter than LTE's[1]. For a 10MHz WiFi channel, Atomix11a achieves a frame decode latency of 36.4μs with a variance of 1.5μs.

We also evaluate the modularity of apps developed in Atomix by developing two apps on top of Atomix11a: (1) we customize Atomix11a to support long-distance links and (2) we add RF localization to Atomix11a.

2 Background and Design Goals

2.1 App Taxonomy

A typical wireless stack is naturally specified in terms of signal processing blocks, a data flowgraph that composes those blocks, and a state machine that selects appropriate flowgraphs to process incoming samples. For example, an 802.11a baseband receiver (fig. 1) has blocks for OFDM demodulation, channel equalization, constellation-to-bit slicing, and Viterbi decoding. Different blocks are composed into separate data flowgraphs for decoding a 54Mbps sample stream, a 6Mbps sample stream, or the PHY layer header, or for transmitting an ACK. Finally, the flowgraphs are assembled into a receiver state machine that transitions from the header decode state to one of the data decode states, optionally followed by the ACK transmit state.

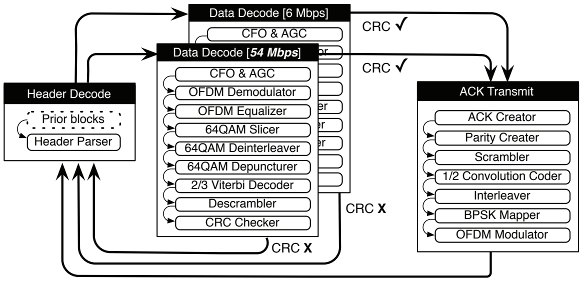


Figure 1: Modules of an 802.11a receiver: blocks, data-flowgraphs, and a state machine.

We classify apps according to the kind of modifications needed to the base wireless stack. (For a wireless stack app that is bootstrapping the base-station, the base stack is null.) To illustrate the classification, we use a simple OFDM receive chain (fig. 2) as the underlying stack that is modified. We classify applications based on modifications that fall into three main categories:

Tap. These modifications tap the signal at various points in the processing chain to implement their functionality. Fig. 2 shows an example of tapping the CSI output and sending it back for offline processing. Such tapping capability is needed for applications like indoor localization [28, 8]. Localization works by converting few samples from the right point in the signal processing chain into location estimates.

Tweak. These modifications tweak parameters of individual signal processing blocks that are already part of the wireless stack to implement tailored functionality. Fig. 2 shows an example of tweaking the channel equalization block. The ability to tweak parameters or modify the functionality of a block can enable applications like better receiver designs (e.g. [13, 25]) for existing transmission formats or adapting the signal chain to a different deployment setting such as long distance rural WiFi links as we show in sec. 5.

Insert (and Delete). These modifications insert new

[1] LTE's 3ms HARQ process turnaround requirement allows for approximately 1ms of decode latency.

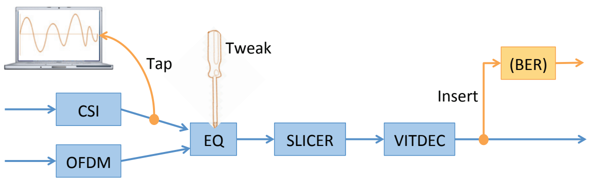


Figure 2: Three basic kinds of modifications

blocks into the signal processing chain, either in the critical path or as additional branches. By inserting new blocks, we can add new functionality to existing signal processing chains. For example, we can add a BER-estimation block for fine-grained channel quality indication to significantly improve rate-adaptation performance [26].

A new wireless stack app that bootstraps a basestation also requires insertion of new blocks.

2.2 Requirements and Design Goals

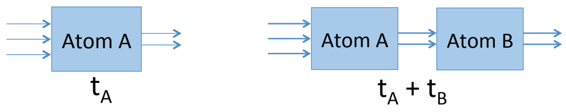
Our goal is to design a software framework for wireless infrastructure in which new signal processing applications can be easily created and deployed. This requires three properties from the framework:

1. Modularity — For a programmer to easily add new apps, the framework must provide a modular interface supporting tap, tweak, and insert primitives. In using those primitives, the programmer must be able to make local code changes without needing to understand or refactor unrelated parts of the existing implementation.

2. Predictable latency — When a new app is deployed, the underlying communication stack should exhibit hardware-like predictable latency. The programmer should be able to precisely estimate and control the latency down to the order of a microsecond. This will let the programmer ensure that the timing requirements specified by the communication protocol are still met (e.g., 16μs sample processing latency for 20MHz WiFi).

3. High computational throughput — As new apps are deployed, the underlying communication stack should remain computationally efficient. The programmer should have the tools needed to utilize the full computational power of the hardware platform to meet sample-rate processing requirements (e.g., 40Msps for a 20MHz WiFi channel). Modularity and predictable latency should not come at the expense of computational efficiency.

While is easy to meet any of these requirements in isolation, combining them generally leads to tradeoffs.



(a) An atom is a unit of execution with fixed timing

(b) Atom compositions have fixed timing

Figure 3: The atom abstraction

In a modular system designed for predictability, modules could always be provisioned for worst-case execution times. However, that would an inefficient choice if modules had large gaps between worst and average-case execution times. Another system that dynamically schedules modules with the objective of keeping processors busy would achieve high average processing efficiency. However, it could cause high variability in execution latencies of modules [15]. By programming an application using low-level instructions, a programmer could tightly control processing latencies; by optimizing it as a monolithic piece of software, she could extract the best possible performance from the hardware. However, the resulting implementation would lack modularity.

These tradeoffs make the design of Atomix challenging. However, as we discuss in the next section, our insights about the application structure of wireless dataplanes allows us to design for all of these goals simultaneously.

3 Design

3.1 The atom abstraction

The design of Atomix is based on the key abstraction of an *atom*:

Atom: A unit of execution with fixed timing.

In Atomix, every operation from signal processing to system handling can be implemented as an atom. Further, those atoms can be composed to form more complex atoms. When atoms are composed, their execution times add up, so that if A and B are atoms, $t(A \circ B) = t(A) + t(B)$ (fig. 3).

The Atomix framework provides the substrate to implement signal processing blocks as atoms (sec. 3.2). It also enables programmers to compose those simple atoms into more complex flowgraph atoms (sec. 3.3) and state atoms (sec. 3.4) to tie together an entire signal processing application. Finally, it provides the capability to map atoms onto multiple cores to create fine-grained pipelines exploiting parallelism (sec. 3.5).

The Atomix framework provides atoms for common platform handling operations such as transferring data between cores and interacting with accelerators. The programmer only needs to implement atoms in C language for custom signal processing blocks. Then, she can tie them together into flowgraphs and a state machine using the language provided by Atomix. The framework provides a high-level declarative API to compose atoms and to map them to multiple cores. Atomix provides a compiler to translate the declarative code into C code. Translated application code and custom signal processing atoms are compiled with the target DSP's native C compiler. Atomix also provides an optimized runtime system (sec. 3.6) to execute compiled atoms efficiently with predictability. The full Atomix application development workflow is described in sec. 3.7.

Block, flowgraph, and state abstractions have one-to-one correspondence with the structure of baseband applications (i.e., the basestation stack and other signal processing applications). Consequently, baseband application software in Atomix has the modularity inherent in the application structure. Since the implementations are entirely made up of atoms, they have predictable timing at every granularity of the modular structure.

To create efficient parallel implementations that can process high sample bandwidths at low latency, the programmer simply ties together appropriate system-handling atoms with signal processing atoms. Then they use the high-level API to map flowgraphs and state machines onto multiple cores. Modularity and predictable timing of atoms simplifies the process of designing pipelines, since the programmer can design a schedule for different bandwidth or latency objectives by simply adding up timings of blocks under various layouts.

In the modular structure of Atomix, signal processing blocks are encapsulated into separate atoms. As a result, atom interfaces provide fine-grained signal tap points; blocks can be tweaked individually by tweaking the corresponding atoms; blocks can be inserted and deleted at fine granularity. Since atoms are tied together using a high-level API, inserting and deleting new atoms is easy.

The timings of atoms add up when they are composed. This simple additive relation allows the programmer to easily infer the effect of a modification on the end-to-end timing of the implementation. On tweaking an atom, she only needs to re-time the tweaked atom and substitute its new cost. On inserting an atom, she simply adds the execution time of the inserted atom to obtain the new end-to-end timing. After predicting the effect of modification on timing, the programmer can easily adjust the multicore layout in the high-level API, if needed to meet processing bandwidth or latency requirements.

We note that the atom abstraction — a unit of execution with *fixed* timing — is an idealized design goal. A software system will inherently have some variability in execution times. For example, the micro-architecture of a processor could affect timing due to bus contention or hardware instruction manipulation. Atomix strives to achieve sufficiently low variability in execution times of atoms for the purpose of building wireless signal processing applications.

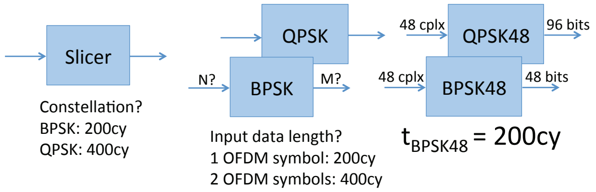


Figure 4: Signal processing blocks to atoms

3.2 Blocks decompose into atoms

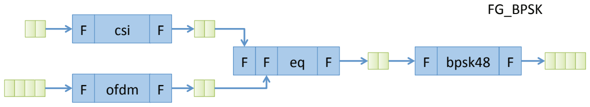
The basic unit of a signal processing application is a signal processing block. A block takes in an input signal and transforms it to another output signal. For example, a constellation bit-slicer block takes in constellation symbols like BPSK or QPSK symbols and produces data bits from them. The execution time of a block depends on the operation it performs, the length of data on which it operates, the processor type (i.e., DSP or accelerator) on which it operates, and the memory location of data buffers it operates on. For a block to be abstracted as an atom, it must execute in known, fixed amount of time. An Atomix signal processing block implements a fixed algorithmic function, operates on fixed data lengths, is associated with a specific processor type, and uses only the memory buffers passed to it during invocation.

Consider the example of a bit-slicing operation, as shown in fig. 4. A slicer block will be decomposed in Atomix as separate BPSK, QPSK, QAM blocks. Each of these blocks will be implemented to accept input and output buffers of fixed lengths, leading to BPSK48 for a BPSK block that takes in 48 complex symbols. The blocks must also be implemented for specific execution core types. If the BPSK48 block could run on both a DSP core and an ARM core, BPSK48DSP and BPSK48ARM would be separate blocks. They may share code internally through common subroutines. By following those decomposition rules, a programmer can implement blocks so that they always executes the same set of instructions on the same kind of processor.

Atomix blocks only make use of memory buffers passed in through function calls, making their memory accesses explicit. Further, the signature of the block implementation is annotated to indicate input and output ports. The Atomix compiler uses I/O port annotations to manage data flow between atoms, as discussed later in sec. 3.3. The Atomix framework provides high-level APIs to control memory placement so that a block al-

Atom	Operation	
F_RG	buffer *	readGet(fifo)
F_WG	buffer *	writeGet(fifo)
F_RD	void	readDone(fifo, buffer)
F_WD	void	writeDone(fifo, buffer)

Table 1: Atomix FIFO atoms (denoted collectively by F).

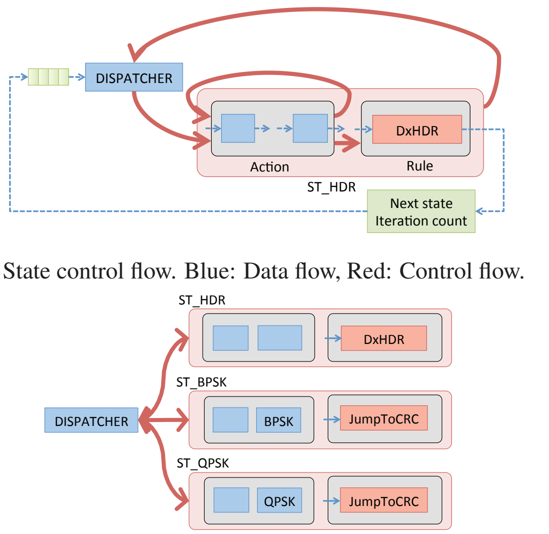


(a) Composing a flowgraph from atoms.

```
#atom:<atomname>:<atomtype>
atom :eq            :EQ
atom :bpsk48        :BPSK48
#fifo:<fifoname>:nbufs=<nbufs>:mem=<memoryid>
fifo :fEqDDSyms     :nbufs=2        :mem=L2
fifo :fDDBits       :nbufs=4        :mem=L2
#flowgraph:<fgname>:<atomname>;+
flowgraph:FG_BPSK:{csi; ofdm; eq; bpsk48;}
#wire:<atomname>:<fifoname>+
wire :eq            :fCSI,fFdDDSyms,fEqDDSyms
wire :bpsk48        :fEqDDSyms,fDDBits
```

(b) Code to compose the flowgraph

Figure 5: Composing a flowgraph from atoms



(a) State control flow. Blue: Data flow, Red: Control flow.

(b) State machine example. Blue: Data flow, Red: Control flow.

```
atom  :dispatcher  :Dispatcher
atom  :dxHDR       :HDRParser
atom  :jumpToCRC   :Jump
fifo  :fDispatcher :nbufs=2    :mem=L2
flowgraph:FG_HDR_AXN:{ ... }
flowgraph:FG_BPSK_AXN:{ ... eq; bpsk48; ... }
flowgraph:FG_QPSK_AXN:{ ... eq; qpsk48; ... }
flowgraph:FG_HDR_RULE:{ dxHDR; }
flowgraph:FG_DD_RULE:{ jumpToCRC; }
wire  :dispatcher  :fDispatcher
wire  :dxHDR       :fDispatcher
#state:<statename>:<FG_action>:<FG_rule>
state :ST_HDR      :FG_HDR_AXN:FG_HDR_RULE
state :ST_BPSK     :FG_BPSK_AXN:FG_DD_RULE
state :ST_QPSK     :FG_QPSK_AXN:FG_DD_RULE
```

(c) Code to compose a state machine

Figure 6: Composing a state machine

ways operates on the same memory types (L2 or DDR etc.). Further, in Atomix, blocks run to completion uninterrupted.

By following simple decomposition rules, Atomix enables the programmer to implement signal processing blocks as atoms. The blocks will run fixed sets of instructions executing uninterrupted on fixed resources using fixed memories. As a result, they will have fixed execution times.

3.3 Flowgraphs are expressed as atoms

The Atomix framework lets the user program application flowgraphs as atoms. For example, the 6Mbps data decoding flowgraph, shown previously in fig. 1, can be expressed as an atom. Flowgraphs are created by tying together signal processing blocks to FIFO queues. FIFOs provide intermediate storage and pass data between signal processing atoms. The framework provides operations to access FIFOs as atoms. This makes an Atomix flowgraph a composition of signal processing atoms and FIFO access atoms. Since a flowgraph is a composition of atoms, it is also an atom in itself.

To program a flowgraph in Atomix, the user declares atoms, FIFOs, wirings between atoms and FIFOs, and the execution sequence of atoms in the flowgraph. Consider the simple signal processing flowgraph shown previously in fig. 2. It shows a channel state information block (CSI) and an OFDM demodulator block (OFDM) both feeding data to a channel equalizer block (EQ) that feeds data to a slicer block (SLICER). A specific realization of this flowgraph with the BPSK48 block is shown in fig. 5a.

In the implementation, block-level atoms and FIFOs are declared with the atom and fifo API, as shown in 5b. In declaring FIFOs, the programmer is able to control the memory region in which the FIFO will be allocated, thus also controlling the execution time of the atoms that will operate on those FIFOs.

The programmer creates a named flowgraph construct that specifies the sequence in which atoms of the flowgraph will be executed (e.g., FG_BPSK in fig. 5b). FIFOs are wired to atoms using the wire API to specify the flow of data between atoms. FIFOs are wired to respective input and output ports of the atoms. Based on the wiring information, the Atomix compiler inserts appropriate FIFO atoms (denoted by F, described in table 1) for each wired atom. FIFO atoms draw and return buffers from FIFO queues. These FIFO access atoms have known, fixed execution times.

Atomix implements a simple control flow model where a flowgraph is executed by executing each of its atoms in sequence without interruption. This straight-through execution model implies that the timing of a flowgraph is simply the sum of timings of its constituent atoms, namely, the signal processing atoms created by the user and the FIFO atoms inserted by the framework.

3.4 States are expressed as atoms

At the top level, Atomix applications are state machines. With blocks and flowgraphs implemented as atoms, the final application component that the programmer needs to create is the state machine. Atomix enables the user to program states as atoms.

The main components of a state machine are a dispatcher atom and named state structures, as shown in fig. 6a. A state is made up of two flowgraphs, the *action* flowgraph and the *rule* flowgraph. Control flow starts from the dispatcher atom which invokes the next state in the state machine's transition sequence with an iteration count `n`. When the framework invokes a state, it executes the action flowgraph `n` times followed by the rule flowgraph once. The action typically performs signal processing operations while the rule flowgraph uses the output of the action flowgraph to decide the next state to transition to.

As an example, consider the reference 802.11a receiver previously shown in fig. 1. An implementation of a subset of that state machine in Atomix is shown in fig. 6b. It shows a header decode state, two data decode states and an ACK transmit state. States are declared with the `state` API as shown in table 6c for reference state machine.

A block making state transition decision writes out a decision buffer into the dispatcher's queue, where the decision buffer indicates the next state and the corresponding iteration count `n`. In the example in fig. 6a, the decision atom `DxHDR` translates the header field decoded by the corresponding action flowgraph into a decision output of (< `ST_BPSK` | `ST_QPSK` >, `n`). When control returns to the dispatcher at the end of `ST_HDR`, the dispatcher reads the next-in-queue decision buffer to continue state transitions.

The components that make up the execution sequence of a state are the dispatcher atom and the action and rule flowgraphs. Since a state is composed of atoms, it is also an atom in itself.

3.5 Atoms generalize to multiple cores

DSPs are able to processing high-bandwidth communication applications at low power because of the parallelism of multiple DSP cores, and hardware acceleration. Atomix enables the programmer to easily parallelize and accelerate flowgraphs and states on multicore DSPs.

Multicore flowgraphs. Flowgraphs are parallelized by splitting into smaller flowgraphs, one for each core. Each sub-flowgraph contains a subset of the blocks in the original flowgraph. The programmer may choose any assignment of blocks to the sub-flowgraphs.

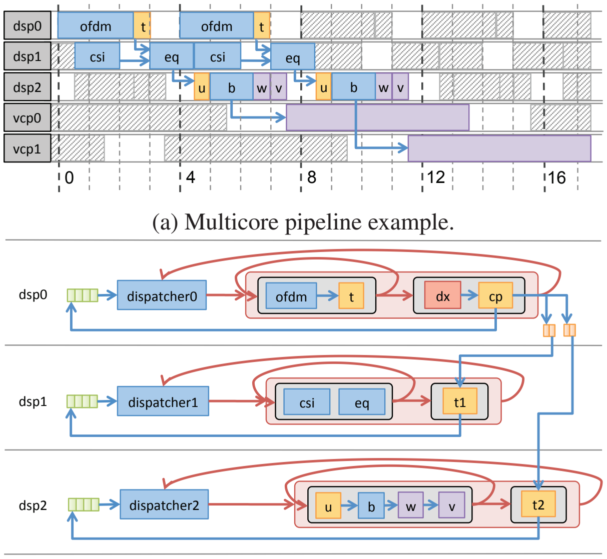
Consider the example of the BPSK flowgraph shown previously in fig. 5a. It is shown laid out as a pipeline on three DSP cores in fig. 7a. The single flowgraph (`FG_BPSK`) is split into three sub-flowgraphs, one for each core: `FGBa0` processes the `ofdm` block on dsp0, `FGBa1` processes the `csi` and `eq` blocks on dsp1, and `FGBa2` processes the `bpsk48` block (shown as `b`) on dsp2. Declarations of these flowgraphs are shown in fig. 7c.

Multicore blocking FIFOs and transfer atoms. When flowgraphs are parallelized, FIFOs are assigned to the same cores as the blocks they are wired to. However, it is possible that a FIFO is wired to multiple blocks that are assigned to different cores. In our example, one such FIFO is at the output of `ofdm` and the input of `eq`. In such a case, the FIFO is replaced with multiple FIFOs, one for each core to which its wired blocks are assigned. The blocks are re-wired to the FIFOs on their own cores to prevent expensive remote FIFO access. This is necessary to preserve the timing of blocks.

When a FIFO is replaced by multiple FIFOs on different cores, data needs to be transferred between them for the flowgraph to compute the correct result. Atomix provides data transfer functionality as transfer atoms. By inserting transfer atoms in the sub-flowgraphs, continuity of the flowgraph is preserved. In the example of `ofdm` and `eq` blocks, the transfer atom `t` is inserted in sub-flowgraph `FGBa0` on dsp0. This atom *pushes* data from the output FIFO of `ofdm` on dsp0 to the input FIFO of `eq` on dsp1. This is shown in figs. 7a and 7b. (The atom `t` could have been inserted into sub-flowgraph `FGBa1` instead. In that case, it would *pull* data from dsp0 to dsp1, and the cost of data transfer operation would be added to `FGBa1`.)

The FIFO read and write atoms are implemented as blocking operations; a read call on a FIFO returns only when it has a filled buffer and, similarly, a write call returns when the FIFO has an empty buffer. By being blocking, FIFO access atoms are able to synchronize execution of multiple cores on data dependencies. For example, the `eq` atom on dsp1 is able to run only when the `ofdm` atom on dsp0 has produced a data buffer and `t` has finished transferring it to dsp1.

Simple extensions to the declarative API let the programmer specify multicore assignment of atoms and FIFOs, as shown fig. 7c. Transfer atoms are inserted into flowgraphs like any other signal processing atoms.



(a) Multicore pipeline example.

(b) Multicore state machine structure.

```
#atom :<name>    :<atomtype>    :core=<id>
atom  :dis0      :Dispatcher    :core=0
atom  :dis1      :Dispatcher    :core=1
atom  :t         :TR            :core=0
atom  :dx        :Jump          :core=0
atom  :v         :VCPIssue      :core=2
atom  :w         :VCPWait       :core=2
#fifo :<name>    :nbufs=<nbufs>:mem=<id>
fifo  :fDis0     :nbufs=4:mem=core0.L2
fifo  :fDis1     :nbufs=4:mem=core1.L2

flowgraph:FGBa0:{ofdm; t;}
flowgraph:FGBr0:{dx; cp;}
flowgraph:FGBa1:{csi; eq;}
flowgraph:FGBr1:{t1;}
flowgraph:FGBa2:{u; b; w; v;}
flowgraph:FGBr2:{t2;}
#state:<stname>:core=<id>:<FGaxn>:<FGrule>
state :ST_BPSK   :core=0:FGBa0:FGBr0
state :ST_BPSK   :core=1:FGBa1:FGBr1
state :ST_BPSK   :core=2:FGBa2:FGBr2
```

(c) Code to compose a multicore pipeline. Highlighted fields are multicore extensions to the API.

Figure 7: Parallelizing atoms on multiple cores.

Multicore states. Multicore states enable parallelized execution of the top-level application state machine. A multi-core state structure is made up of a pair of action and rule flowgraphs for each core. When the multicore system enters a state, each core executes its respective flowgraph pair for that state. By setting the core-specific actions of a multicore state to sub-flowgraphs of a parallelized flowgraph, the programmer is able to create efficient multicore processing pipelines.

A three-core version of the BPSK state `ST_BPSK` is shown in fig. 7b. For this state, the cores now use sub-flowgraphs `FGBa0`, `FGBa1`, and `FGBa2` as actions. Similarly, they use `FGBr0`, `FGBr1`, and `FGBr2` for rules. The code to declare the multicore state is shown in fig. 7c.

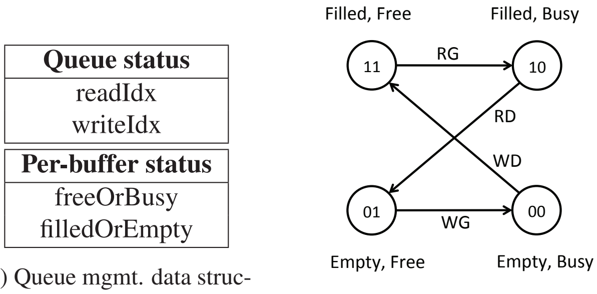
A multicore state machine executes like multiple parallel state machines executing asynchronously. Each core has its own dispatcher atom (fig. 7b. On any core, control flow starts at the dispatcher, passes through the next state to execute, and returns to the dispatcher. Cores synchronize on the state transition sequence by exchanging transition decisions computed in the rule flowgraphs.

In our example, once the system enters `ST_BPSK`, all cores start processing their respective action flowgraphs for the state. These sub-flowgraphs exchange data through the transfer atoms that the programmer inserted when parallelizing them. Collectively, they execute the parallelized BPSK processing pipeline.

After finishing a state's action iterations, cores independently execute their respective rule flowgraphs for that state. In our configuration, only dsp0 executes the decision atom `dx` for the BPSK state. Its output decision is distributed to each dispatcher through transfer atoms `t1` and `t2`. By executing state-transition decision atoms on a single (though possibly different) core for each state, the programmer synchronizes state transitions across all cores. In this way, the entire system transitions states in lock-step.

Accelerator atoms. The pipeline shown in fig. 7a includes atoms for Viterbi-decoding on Viterbi Co-Processors (VCPs). To use the VCPs, Atomix provides VCPIssue (`v`) and VCPWait (`w`) atoms. A VCPIssue atom can configure a VCP and start its execution. A VCPWait atom can wait for VCP execution to terminate and thus synchronize a DSP core with a VCP core. A hardware accelerator takes a fixed amounts of time to execute a given workload. Consequently, issue and wait operations interacting with an accelerator finish executing in predictable amounts of time, making them atoms. This model extends to other accelerators.

Multicore execution model and timing. The multicore execution model of Atomix is similar to the single-core setting. Blocks access FIFOs for input and output buffers before they execute, they run as soon as buffers are available, and they always runs to completion. Cores communicate by transferring data between FIFOs, and syn-



(a) Queue mgmt. data structures

(b) Buffer state machine

Figure 8: Lock-free queue management

chronize execution by polling FIFOs with blocking calls. When a block polls a FIFO, its wait duration is determined by execution times of upstream atoms, including data producers and transfer atoms.

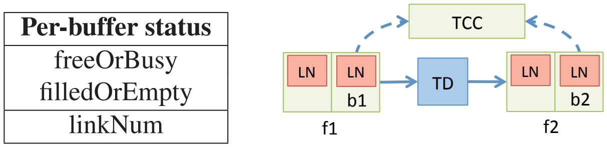
In this manner, execution control flow simply mimics data flow in the system. This allows the timing of a multicore atom to be computed by adding up the execution times of atoms on the critical path of execution, i.e., the slowest path of data flow in the parallel execution.

To illustrate the timing model, we analyze the BPSK action flowgraph pipeline. We assume that data is arriving every 4 units of time. We also assume that the blocks have the following execution costs per iteration: `ofdm`:2.5, `csi,eq`:1.5, `b`:1.0, `t,u,w,v`:0.5, and VCP-processing:6.0. The total DSP computation load of this flowgraph is 6.0 units, which cannot be sustained at data arrival rate by a single DSP. When laid out on three DSP cores and two VCPs with transfer atoms, the pipeline is able to meet the processing throughput requirement, as shown in fig. 7a. The pipeline's latency is the timing of the critical path `ofdm-t-eq-u-b-w-v-VCP`, which is 13.5 units.

3.6 Efficient data-flow implementation

The runtime system of Atomix executes fine-grained pipelines with hardware-like efficiency using two main techniques: lock-free FIFOs, and asynchronous data transfers.

Lock-free FIFO implementation. The FIFO implementation in Atomix provides a simple API with four functions (table 1). We design the functions to execute extremely efficiently: the `Get` functions take 40 cycles each, and the `Done` functions take 8 cycles each. Typically, FIFO queues in multicore environments are implemented with locks to serialize access. Atomix does use them because locks are expensive, and they can create timing variability by serializing concurrent accesses in arbitrary order. Atomix FIFO API implementation is entirely lock-free.



(a) Extended buffer state

(b) Transfer scenario

Figure 9: Link number field and Transfer Completion Code (TCC) for handling asynchronous transfers

In order to operate correctly without locks, Atomix requires every FIFO to have a single-reader and a single-writer (SRSW) at any point in multicore execution. In addition to common readIdx and writeIdx status for the queue, a per-buffer 2-bit (freeOrBusy, filledOrEmpty) status tuple is maintained (fig. 8a). As FIFO API calls are made, the buffer transitions through those states (fig. 8b). If a call cannot succeed (e.g., RG in state 01), it blocks until the buffer reaches the required state for the call to proceed. Only one of the four possible API calls can succeed and hence, modify the data structures in any given state. By the SRSW property, only one FIFO accessor will ever be allowed to write to the FIFO data structure, ensuring race-free operation without locks.

The SRSW constraint may seem too restrictive compared to typical FIFO APIs. However, in our experience, most FIFOs wireless applications naturally have single readers and single writers. Multiple readers or writers from different states could still be wired to the same FIFO since at any given time, the system is in exactly one state.

Handling asynchronous transfers. The FIFO functions `read/writeDone` cannot be used with asynchronous DMA transfer. In order to run them upon DMA transfer completion, DSP cores must be interrupted, which would cause variability in atom execution. To deal with this issue, we introduce the buffer state forwarding mechanism, where the FIFO manager is able to deduce `readDones` and `writeDones` without those calls being made explicitly.

To implement buffer state forwarding, we extend per-buffer status field with a linkNum field (LN). On our prototyping platform, TI KeyStone DSPs, DMA channels have associated event registers to indicate transfer completion. We denote this register by TCC (fig. 9). When the DMA transfer atom `TD` issues a DMA request, it sets up the LN field of source and destination buffers to point to the TCC register of the DMA channel used for the transfer. `TD` issues the request and returns. The core that executed `TD` moves on to other atoms. The transfer would finish later asynchronously but the buffers will continue to be marked busy. However, when the buffers are accessed again in FIFO order, the queue manager is able to identify from non-zero LN field that they were marked busy for an asynchronous transfer. The queue manager then polls the TCC flag pointed by LN. When the TCC indicates transfer completion, the queue manager forwards the state of the buffer as if the corresponding `readDone` or `writeDone` was called on the buffer.

3.7 Full app development workflow

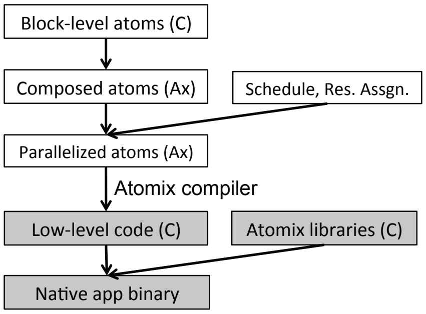


Figure 10: Full Atomix app development workflow

Stages in Atomix app development. To summarize the Atomix framework design, the following is an overview of the application development workflow (fig. 10). First, the user implements signal processing blocks as atoms in C language. Next, she composes blocks into flowgraphs and states using the declarative interface of Atomix. Then, she computes a parallelized schedule and resource assignment that will meet the latency and throughput requirements of the application, and incorporates it in the declarative app code. This high-level app code is then compiled down to low-level C code by the Atomix compiler. Finally, the low-level application C code is compiled with the platform's native C compiler and linked against Atomix runtime libraries into a binary. The modular structure of Atomix applications and the streamlined development flow let the user rapidly iterate designs and optimize performance.

Algorithmic scheduling. The execution model of Atomix makes it possible to algorithmically compute the best parallelized schedule for an application. Blocks can be profiled individually for execution times. Then, finding the optimal schedule for a flowgraph can be expressed as a resource assignment problem with dependency constraints. FIFO access costs, data transfer costs and accelerator access costs can all be modeled as additional constraints. We have been able to formulate the flowgraph scheduling problem as an Integer Linear Program (ILP). The flowgraph scheduling problem in Atomix is similar to the instruction-loop scheduling problem solved by VLIW compilers [17, 18]. Incorporating algorithmic scheduling in the Atomix compiler can simplify the application development flow even further.

4 Building an 802.11a Receiver in Atomix

We put Atomix framework's capability of providing low latency and high throughput to test by using it to implement a 10MHz 802.11a receiver called Atomix11a. We implement 802.11a as a benchmark due to its particularly demanding requirements. It uses the same bandwidth modes as LTE (5, 10, 20 MHz), imposing similar processing complexity. However, it places much more stringent frame-decoding latency constraints than LTE - 64μs for 5MHz and 32μs for 10MHz compared to more than 1ms for LTE. To stress the system, we implement the highest throughput Modulation Coding Scheme (MCS) in 802.11a, MCS 7: 64-QAM with 3/4 coding rate, which operates at 27Mbps on a 10MHz channel.

We first evaluate Atomix11a and show that (1) Atomix11a can decode over-the-air frames in real-time on the TI 6670 a 4-core, 4-Viterbi accelerator 1.25GHz DSP, (2) Atomix11a can achieve 36μs processing latency with 1.5μs of variability, sufficient to meet requirements of a 5MHz 802.11a channel and close to meeting 10MHz requirements, and (3) most of the atoms we built for Atomix11a have predictable timing. Next, we describe the use of Atomix to implement a modular 802.11a receiver that has low latency and high throughput on the low power (7 Watt) TI 6670 DSP. We could implement the receiver in about 3000 lines of declarative application code (excluding individual block implementations in C language).

4.1 Evaluation of Atomix11a

We first demonstrate that Atomix11a is a robust, faithful implementation of 802.11a signal processing. Then we evaluate Atomix11a's latency and timing variability, and finally we evaluate the runtime of Atomix11a's atoms to show that they come close to Atomix's fixed runtime atom abstraction.

Atomix11a exceeds receiver sensitivity requirements. An 802.11a compliant receiver must be able to decode 1,000 byte packets with a Packet Error Rate (PER) of 0.10 at -65dBm receive power over an AWGN channel, as measured at the RF frontend's antenna port [7]. In our experiment, we feed signal over an RF co-axial cable to emulate an AWGN channel without multipath reflections. On the receiver, we use the high-quality RF frontend of an R&S FSW spectrum analyzer to digitize received signal into baseband I/Q samples, so the experiment focuses on the robustness of Atomix11a

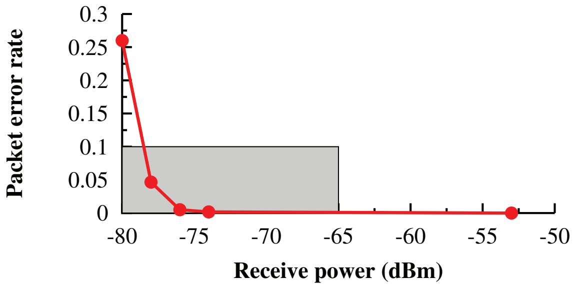


Figure 11: Atomix11a exceeds receiver sensitivity specifications (gray box).

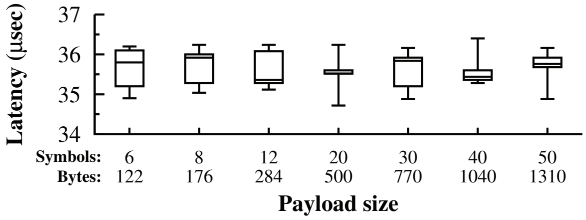


Figure 12: Atomix11a processing latency is at most 36.4μs and it varies at most by 1.5μs.

baseband running on the TI 6670 DSP, not the quality of the RF frontend. Fig. 11 shows the results of this experiment. Atomix11a exceeds 802.11a's requirement of 0.10 PER at -65dBm; it achieves a 0.046 PER at -78dBm.

Atomix11a operates robustly indoors. The second robustness test is to see if Atomix11a can decode packets sent over-the-air in challenging indoor multipath channels causing frequency selective fading. Robust 802.11a receivers use a computationally intensive zero-forcing channel equalizer to equalize the sub carriers.

We setup a 6 meter link across an office and transmitted 100,000 802.11a frames, each 1,000 bytes, at MCS 7 (64-QAM, 3/4 coding rate), and over the 10MHz channel at 2.479GHz. We used a USRP2 RF frontend connected to the TI 6670 DSP with gigabit ethernet. Both the transmitter and the receiver were connected with 3 dBi omnidirectional antennas. The receiver successfully decoded (verified CRC) for 99,999/100,000 frames. Therefore Atomix11a is capable of an extremely low PER of 0.001% in an indoor office environment, and is likely a faithful implementation of 802.11a.

Atomix11a has low processing latency. Next, we perform an end-to-end test of Atomix framework's ability to support a low latency, low timing variability, and high throughput signal processing chain. We transmitted 200 frames each of different sizes (6-50 symbols, 122-1310 bytes) to the USRP2 which is connected to the TI 6670 DSP. We compute the processing latency by subtracting the frame processing time of Atomix11a from the frame's airtime. We measure Atomix11a's maximum processing latency, as well as the range over which it varies.

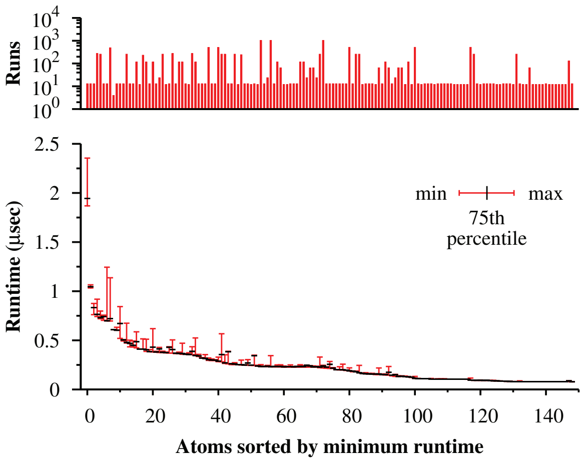


Figure 13: Atomix11a atoms have low timing variability.

Fig. 12 shows the results of this experiment. The whiskers are the minimum and max of each of the 200 frames, and the boxes show the 25th, 50th, 75th percentiles. For all the frame sizes tested, the minimum and maximum processing latency is similar, indicating that the composition of atoms adds minimal latency between frames. The max processing latency is 36.4μs, 1.14× the requirement of 32μs for 802.11a. Although 36.4μs latency is low, the Atomix11a implementation could be further optimized to meet protocol latency requirements. Specifically, there is room to optimize the implementations of individual atoms, and to algorithmically compute the parallelization schedule of atoms that minimizes decode latency.

Most Atomix11a atoms have low timing variability. In the final experiment, we observe the variability of the runtime of every atom in Atomix11a. We expect some variability due to L1 caching and micro-architectural sources of variability like bus contention (sec. 3.2). We instrumented each of the atoms in the Atomix11a implementation with a lightweight cycle counter that records the cycle count of each execution of the atom. We measured the runtime of every atom that executes while receiving 12 frames of 500 bytes (20 OFDM symbols at MCS 7). Most atoms run every OFDM symbol of the frame.

Fig. 13 shows the min, max, and 75th percentile runtime of all of the 150 atoms that were executed in Atomix11a (bottom), as well as the number of times each of those atoms were run in our experiment (top). Most

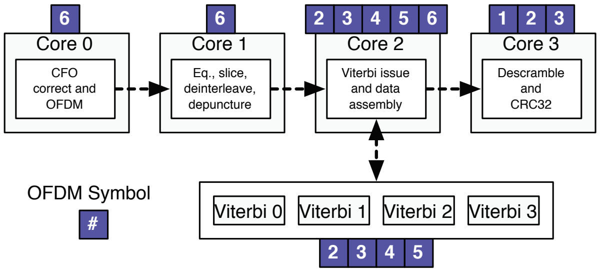


Figure 14: Fine-grained OFDM symbol pipelining in Atomix11a's data decode state.

atoms have a fixed or insignificant runtime (atoms 60-150). These atoms include both framework atoms such as memory transfers, as well as Atomix11a computation atoms such as the 64-QAM soft slicer and the PLCP's soft deinterleaver.

Atomix11a's atoms have at most a 0.486μs range between their max and min runtime due to L1 caching and micro-architecture. The atom with the largest runtime range (atom 0) is the channel estimator, which is the most computationally intensive atom in Atomix11a. Although the difference between the max and min runtime for these atoms can be significant, for all atoms the 75th percentile of runtime is close to the minimum. This shows that the atom abstraction holds well in practice. Further, execution times of different atoms have low co-variance. This explains the low end-to-end packet decoding variability of 1.5μs. This variability can be reduced further by disabling L1 caching on the TI 6670 and managing the L1 SRAM in software with Atomix transfer atoms.

4.2 Implementation of Atomix11a

Modularity resulting from the atom abstraction enabled us to implement Atomix11a with fine-grained pipeline parallelism, which was crucial in achieving high through and low latency on the TI 6670 DSP. We implemented signal processing blocks as fine-grained atoms that operate on one OFDM symbol (8μs worth of samples at 10MHz) every iteration, as discussed earlier. The Atomix API enabled us to compose the block-level atoms into flowgraphs and states and schedule them for high performance in a low power budget of 7W.

Precise timing of atoms allowed us to spread the pipeline over all the cores and accelerators so it would execute without stalls and achieve the required processing throughput of 10Msps reliably. Fine-grained atoms allowed us to pipeline data processing with data reception to achieve low decode latency of 36μs.

Fine-grained pipeline structure. The fine-grained pipeline structure of Atomix11a is shown in fig. 14. It depicts a snapshot of the pipelining in payload decoding state. At any point in the steady state, six OFDM symbols are being processed in the pipeline. With fine-grained resource allocation API of Atomix, we could lay out the pipeline so that each DSP core and Viterbi co-processor (see ahead) contributed to the processing of an OFDM symbol. This enabled the highest sample processing throughput achievable by the hardware resources.

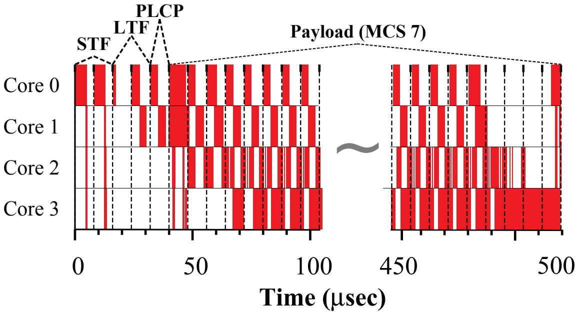


Figure 15: Core utilization of Atomix11a while it receives a 1500 byte MCS 7 frame at 10MHz.

The core utilization map resulting from our fine-grained pipeline is shown in fig. 15. It depicts the processor's active and idle time on all four DSP cores while it is receiving a 1500 byte 802.11a MCS 7 frame. The vertical grid indicates the arrival of a symbol to Atomix11a. As the figure shows, all cores are participating in a pipeline to process every arriving OFDM symbol.

Parallelizing Viterbi decoding for high throughput. Optimal Viterbi-decoding is a sequential algorithm. However, in practice, the decoded sequence of bits (codeword) can be partitioned into overlapping chunks that can be decoded in parallel with negligible performance penalty under certain overlap size constraints. We use this scheme to use all 4 VCPs for decoding the same WiFi frame, where each chunk is one-half OFDM symbol with appropriate padding of surrounding bits to satisfy overlap constraints. Such a scheme has also been used in BigStation [30] where it is described in more detail.

Optimizing latency with overlapping states. Atomix allows different cores to operate simultaneously in different states. This feature enables the implementer to use all cores to execute as many states at once as possible, and thus provide lower latency than if the states ran in serial. For example, fig. 15 shows as the LTF symbol arrives we split its processing across two cores, and same for the PLCP symbol[2]. The split point for the LTF

[2] Note that with a 10MHz signal, the LTF and PLCP processing ends before the arrival of the next symbol. However, with a 20MHz signal

state is after CFO estimation because the CFO estimates can immediately be applied on the PLCP symbol when it arrives. Then, when the channel estimates finish on core 1 in LTF, the rest of the PLCP is processed. Core 1 uses the channel estimates to complete the PLCP processing with the symbol transferred from core 0. In summary, Atomix enabled pipelined processing of two symbols in two states at the same time on two cores.

Easy to program. We implemented the Atomix11a receiver in 3000 lines of Atomix's high-level code. The Atomix compiler translated the high-level code into about 30,000 lines of low-level application-specific C code. Atomix reduces the LoC effort by an order-of-magnitude. Further, it makes the development process significantly easier through its abstractions compared to directly writing low-level code. All of the blocks in Atomix11a's signal processing library were implemented with less than 300 lines of C code each.

Low resource footprint. The Atomix atom abstraction requires each configuration of a signal processing block to be split out as a separate block, e.g. `BPSK48`, `BPSK96`, `QPSK48`. To avoid code size explosion, we implement similar blocks as wrappers on a common parameterized kernel (e.g., `BPSK<N>`). Atomix API also provides primitives to instantiate parameterized blocks in the atom declarations.

Using those techniques, Atomix11a has code size of 468 KByte per core. This comfortably fits in the 512 KByte of L2 SRAM per core that we allocated as program memory. The data size on any core is at most 145 KByte out of 512 KByte of L2 SRAM allocated for data memory. Code size could be further optimized by creating per-core binaries containing only code executed on each of the cores. Another approach is to write the blocks in C++ using templates. It is also possible to partition the SRAM in favor of program memory.

5 Modifiability

We demonstrate the modularity of the Atomix 802.11a receiver by adapting it to two new applications: long-distance links, and phase-array location signatures.

5.1 Adapting to long-distance links

Long-distance wireless links appear in settings like rural connectivity [6], point-to-point backhaul and community networks [4]. They are challenging because the *delay spread* of the multipath wireless channel also grows with link distance. However, WiFi is designed for shorter distance links. As a case-study of Atomix modifiability,

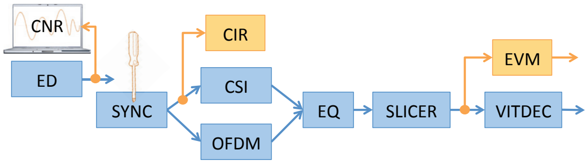
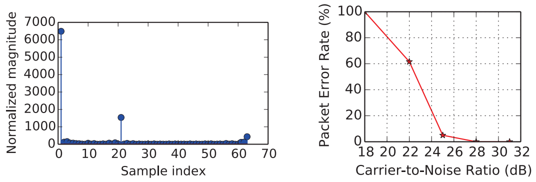


Figure 16: Modifications for long-distance app



(a) Measured CIR (b) PER with long-CP

Figure 17: Performance on a two-tap multipath channel. Long cyclic-prefix (CP) is critical for long outdoor links.

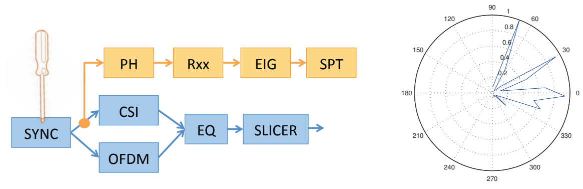
we demonstrate the process of systematically diagnosing and adapting our base WiFi receiver to work well over an emulated long-distance link.

We emulate a long-distance channel that has two multipath components: the line-of-sight component, and a reflection delayed by 2us (20 samples) that is 12dB lower in strength. On this channel, the unmodified Atomix11a receiver resulted in 100% packet error rate (PER). To ascertain that the signal strength is not the limiting factor, we tap the baseband chain to read off sample energy values being computed by the energy-detection block (ED), as shown in fig. 16. On the simulated transmission with an average carrier-to-noise ratio (CNR) of 45dB, our tap revealed the CNR to be 45.4dB, confirming sufficient signal strength for successful decode.

Ruling out signal strength limitation, we suspected a high-distortion channel. To look deeper, we inserted an error-vector-magnitude (EVM) computation block after slicing the constellation. On a distortion channel, we would expect a high EVM value, or equivalently, a low SNR indicated by the EVM. The EVM block runs in 0.64us on core 1. We set it to compute on the output of the PLCP field. Core 1 had enough cycles (fig. 15) in the PLCP decode state to run EVM without adding latency to packet decode. Our EVM block indicated only 19.2dB SNR, much lower than 45.4dB CNR. This strengthened the suspicion of a poor channel.

To find the ground truth about the channel, we inserted a block to compute the time-domain channel impulse response (CIR) using the long training field (LTF) of the preamble, same as used for frequency-domain channel estimation. The CIR block runs in 3.4μs. We set it to run core 3, which has enough spare cycles after packet detection to run CIR without hurting packet decode. The

(4μs interval), the PLCP symbol will arrive during LTF processing.



(a) Modifications for AoA app

(b) AoA spectrum

Figure 18: AoA spectrum app

CIR block showed the channel response to be as shown in fig. 17a. It accurately recovered the emulated channel and pointed us to the root problem: two strong components 20 samples apart, and a delay spread too high for the standard WiFi OFDM cyclic prefix length of 16 samples.

Armed with this knowledge, we implemented the solution of increasing the cyclic prefix (CP) length to 32 samples. To implement the longer CP communication mode, only the SYNC block needed tweaking. It is responsible for discarding the cyclic prefix from each OFDM symbol before passing it to rest of the processing chain. We tweaked it to discard 32 samples instead of 16. After implementing the longer CP, performance of the link came back to the expected CNR-PER quality on a benign channel. The observed performance of CP length 32 over the 2us two-tap channel is shown in fig. 17b. On the same reception as before, the EVM block now indicated SNR of 38.7dB, much closer to the CNR of 45.4dB, confirming that signal distortion had been effectively corrected.

By simply tapping existing signals, tweaking a block and inserting a few blocks, we could methodically adapt the receiver to a new application setting. The whole exercise took just a few hours of programmer-time to modify or add a total of 20 lines of declarative code (after we implemented the individual signal processing blocks).

5.2 Adding location signatures

Location signatures of a wireless transmitter can be computed using phase-array processing at a MIMO AP receiver. MUSIC [19] is a popular algorithm to compute angle-of-arrival (AoA) spectrum which is used in many location-based systems like [28, 29]. It is desirable to embed AoA spectrum computation on the AP to save bandwidth incurred in remote computation, and to make location estimates available at the AP at sub-millisecond latencies. For such scenarios, we demonstrate addition of an AoA computation application to our Atomix 802.11a receiver.

The set of blocks used to compute AoA spectrum is shown in fig. 18a. A flowgraph is composed with the phase-multiplier block (PH), correlation matrix computation block (Rxx), a complex Hermitian eigendecomposition block (EIG), and a spectrum computation block (SPT). We leverage the baseband receiver to detect WiFi packets using preamble-detection. Preamble samples are tapped and copied on a FIFO for AoA computation. Once the packet decode finishes, AoA computation is invoked. A sample of AoA spectrum computed on the DSP processor is shown in fig. 18b.

The entire AoA computation chain takes 530us with room for further optimization. Its components are: PH: 1.5us, Rxx: 21us, EIG: 178us and SPT: 330us. For eigendecomposition, we use a FORTRAN-to-C translated routine from EISPACK [2]. Our simple implementation is already able to compute an AoA spectrum in about half of a millisecond at the base-station. We are able to reproduce the results from [28, 29], which we omit for brevity. To add the AoA app, we needed to add only about 30 lines of declarative code to the base Atomix11a receiver (in addition to blocks in C).

6 Related work

Existing frameworks suited to building baseband applications using abstractions of blocks and flowgraphs have only targeted GPPs and FPGAs. Also, operating systems for building realtime programs on DSPs do not provide the right abstractions for modular communication applications. To the best of our knowledge, Atomix is the first system to enable high-performance, modular communication applications on multi-core DSPs.

Modular frameworks for GPPs and FPGAs. GNU Radio [5] is a rapid prototyping toolkit for signal processing applications. It provides interfaces to connect blocks into flowgraphs on GPPs and DSPs [10]. However, unlike Atomix, GNU Radio's abstractions do not have guaranteed timing. They depend on abundant processing resources for real-time performance.

The SORA [21] software radio and its modular software architecture called Brick [11] provide real-time performance of wireless applications on GPPs. With similar modularity goals, SORA/Brick and Atomix share similarities in the programming interfaces. However, the differences in GPP and DSP architectures make for very different designs of the two systems. SORA's design includes mechanisms like binding threads to cores, masking interrupts from cores and special memory handling for optimization cache performance on signal processing apps. These apply well to the GPP environment with a general-purpose operating system but do not carry over to the DSP architecture.

Ziria [20] is a domain specific language targeted to GPPs for implementing efficient PHY designs. It facilitates concise and error-free expression of PHY control logic. It also provides support for automatic vectorization of individual signal processing blocks. The advantages of programming signal processing apps in Ziria, currently limited to GPPs, are complementary to those of Atomix. They can be extended to DSPs by a compiler that translates Ziria programs to Atomix programs, using Atomix as a convenient abstraction layer.

The Click modular router [9] provides an abstraction of blocks and flowgraphs for building packet-based network infrastructure. The expressiveness of Click's simple elements and FIFOs was an inspiration for Atomix. However, Click is not suited to developing wireless applications on DSPs. For instance, Click's elements lack timing guarantees; the Click dynamic scheduler is a source of timing variability.

StreamIt [24] provides a flowgraph model and intuitive syntax to program stream processing applications. The StreamIt programming language explicitly reveals parallelization opportunities to the compiler so it can exploit task, data, and pipelining parallelism. In that sense, StreamIt's compiler complements Atomix. However, StreamIt's abstractions lack guaranteed execution times. Moreover, StreamIt encourages embedding branches within blocks that can cause variability. Atomix forbids this behavior, and requires all branches to be expressed as their own atoms.

AirBlue [12] simplifies design of cross-layer signal processing applications for FPGAs. Like Atomix, AirBlue also requires FIFOs between signal processing blocks to enable modular modifications. However, modular implementation of an application for predictable and efficient execution poses different challenges on multi-core DSPs and FPGAs.

Real-time operating systems for DSPs. Real-time operating systems such as QNX Neutrino [16], WindRiver VxWorks [27] are used in automotive, robotics and avionics systems. DSPs have their own real-time operating systems such as TI SYS/BIOS [22], and 3L Diamond [1]. In principle, the real-time guarantees provided by these systems (e.g., priority-scheduling, predictable interrupt timing) can be applied to real-time wireless infrastructure applications too. However, these abstractions can only be indirectly mapped to the blocks and flowgraphs that make up baseband applications. Real-time OSes generally provide some form of FIFO-based interprocess communication that can be used to create flowgraphs. However, such FIFOs are not designed for the fine-grained pipeline parallelism that Atomix's lock free FIFOs enable.

7 Discussion

Atomix in the development pipeline: Atomix aims to make signal processing applications easy to deploy on DSPs in the wireless infrastructure.However, Atomix can also enable at-scale prototyping and testing of wireless applications. Platforms like USRP E310 and E110 integrate DSPs and ARM cores with commodity RF frontends in embedded form-factors. These platforms are ideal for building wide-scale research testbeds. Atomix makes them easy to program for rapid prototyping.

Limitations of Atomix: Atomix pushes conditionals out of blocks to the top level construct of state machine. This makes stepping through conditionals much slower than having them embedded within blocks. Most baseband apps spend majority of their time in computationally heavy signal processing blocks, making the cost of stepping through states negligible. However, for applications dominated by data-dependent workloads like iterative computations like successive interference cancellation or searching/sorting, Atomix can be inefficient.

8 Conclusion

Atomix is a framework for building and widely deploying modular, predictable latency, high throughput baseband signal processing applications. The key abstraction that enables Atomix is that of an atom — a computation unit with fixed execution time. Atomix demonstrates that by abstracting operations as atoms, ease of programming and fine control for predictable timing can be provided simultaneously without sacrificing efficiency for signal processing applications. We show that it can be used to deploy apps in both WiFi and LTE infrastructures.

We plan to make Atomix available to the research community and enable the implementation and evaluation of new baseband apps at scale. We also plan to further develop Atomix to be a programmable dataplane for a software-defined radio access network. Specifically, we plan to investigate open APIs (analogous to OpenFlow) that programmable basestations should expose to enable networks to tightly manage packet processing on a per-flow basis, and enable software defined control over the wireless infrastructure.

9 Acknowledgments

We thank Kyle Jamieson and Jie Xiong for providing us with a reference implementation of SecureArray [29]. We thank Jeffrey Mehlman for help with implementation, and Yiannis Yiakoumis for suggestions on presenting the content. We also thank our shepherd Deepak Ganesan for his valuable feedback, and the anonymous reviewers for their insightful comments. This work was supported by the NSF POMI grant.

References

[1] 3L Diamond. http://www.3l.com/products/3l-diamond.

[2] EISPACK. http://www.netlib.org/eispack.

[3] 3GPP. Specification Release version matrix. http://www.3gpp.org/specifications/67-releases.

[4] J. Bicket, D. Aguayo, S. Biswas, and R. Morris. Architecture and evaluation of an unplanned 802.11b mesh network. In *Proc. ACM Conference on Mobile Computing and Networking (MobiCom)*, 2005.

[5] E. Blossom. GNU radio: tools for exploring the radio frequency spectrum. *Linux J.*, 2004:4–, June 2004.

[6] K. Chebrolu, B. Raman, and S. Sen. Long-distance 802.11b links: Performance measurements and experience. In *Proc. ACM Conference on Mobile Computing and Networking (MobiCom)*, 2006.

[7] IEEE. 802.11a-1999.

[8] K. Joshi, S. Hong, and S. Katti. Pinpoint: Localizing interfering radios. In *Proc. Symposium on Networked Systems Design and Implementation (NSDI)*, 2013.

[9] E. Kohler, R. Morris, B. Chen, J. Jannotti, and M. F. Kaashoek. The Click modular router. *ACM Transactions on Computer Systems (TOCS)*, 2000.

[10] S. Ma, V. Marojevic, P. Balister, and J. H. Reed. Porting GNU Radio to multicore DSP+ARM system-on-chip – a purely open-source approach. In *Karlsruhe Workshop on Software Radios*, 2014.

[11] Microsoft Research. Brick Specification. *The SORA Project*, 2011.

[12] A. Ng, K. E. Fleming, M. Vutukuru, S. Gross, Arvind, and H. Balakrishnan. Airblue: A system for cross-layer wireless protocol development. In *Proc. ACM/IEEE Symposium on Architectures for Networking and Communications Systems (ANCS)*, 2010.

[13] K. Nikitopoulos, J. Zhou, B. Congdon, and K. Jamieson. Geosphere: Consistently turning MIMO capacity into throughput. In *Proc. ACM SIGCOMM*, 2014.

[14] R. Patra, S. Nedevschi, S. Surana, A. Sheth, L. Subramanian, and E. Brewer. Wildnet: Design and implementation of high performance WiFi based long distance networks. In *Proc. Symposium on Networked Systems Design and Implementation (NSDI)*, 2007.

[15] P. Puschner and A. Burns. Guest editorial: A review of worst-case execution-time analysis. *Real-Time Systems*, 2000.

[16] QNX Software Systems, Ltd, Kanata, Ontatio, Canada. http://www.qnx.com.

[17] B. R. Rau. Iterative modulo scheduling: An algorithm for software pipelining loops. In *Proc. International Symposium on Microarchitecture (MICRO)*, 1994.

[18] J. Sánchez and A. González. The effectiveness of loop unrolling for modulo scheduling in clustered VLIW architectures. In *Proc. International Conference on Parallel Processing (ICPP)*, 2000.

[19] R. O. Schmidt. Multiple emitter location and signal parameter estimation. *IEEE Transactions on Antennas and Propagation*, 1986.

[20] G. Stewart, M. Gowda, G. Mainland, B. Radunovic, D. Vytiniotis, and C. L. Agulló. Ziria: A DSL for wireless systems programming. In *Proc. ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2015.

[21] K. Tan, J. Zhang, J. Fang, H. Liu, Y. Ye, S. Wang, Y. Zhang, H. Wu, W. Wang, and G. M. Voelker. Sora: High performance software radio using general purpose multi-core processors. In *Proc. Symposium on Networked Systems Design and Implementation (NSDI)*, 2009.

[22] Texas Instruments. *SYS/BIOS (TI-RTOS Kernel)*.

[23] Texas Instruments. TMS320C6670 - Multicore Fixed and Floating-Point System-on-Chip. *http://www.ti.com/product/tms320c6670*.

[24] W. Thies, M. Karczmarek, and S. P. Amarasinghe. Streamit: A language for streaming applications. In *Proc. International Conference on Compiler Construction (CC)*, 2002.

[25] A. Vigato, S. Tomasin, L. Vangelista, V. Mignone, N. Benvenuto, and A. Morello. Coded decision directed demodulation for second generation digital video broadcasting standard. *IEEE Transactions on Broadcasting*, 2009.

[26] M. Vutukuru, H. Balakrishnan, and K. Jamieson. Cross-layer wireless bit rate adaptation. In *Proc. ACM SIGCOMM*, 2009.

[27] Wind River Systems, Inc, Alameda, CA, USA. `http://www.vxworks.com`.

[28] J. Xiong and K. Jamieson. Arraytrack: A fine-grained indoor location system. In *Proc. Symposium on Networked Systems Design and Implementation (NSDI)*, 2013.

[29] J. Xiong and K. Jamieson. Securearray: Improving WiFi security with fine-grained physical-layer information. In *Proc. ACM Conference on Mobile Computing and Networking (MobiCom)*, 2013.

[30] Q. Yang, X. Li, H. Yao, J. Fang, K. Tan, W. Hu, J. Zhang, and Y. Zhang. BigStation: Enabling scalable real-time signal processingin large MU-MIMO systems. In *Proc. ACM SIGCOMM*, 2013.

WiDeo: Fine-grained Device-free Motion Tracing using RF Backscatter

Kiran Joshi, Dinesh Bharadia , Manikanta Kotaru, Sachin Katti
krjoshi, dineshb, mkotaru, skatti@stanford.edu

Abstract

Could we build a motion tracing camera using wireless communication signals as the light source? This paper shows we can, we present the design and implementation of WiDeo, a novel system that enables accurate, high resolution, device free human motion tracing in indoor environments using WiFi signals and compact WiFi radios. The insight behind WiDeo is to mine the backscatter reflections from the environment that WiFi transmissions naturally produce to trace where reflecting objects are located and how they are moving. We invent novel backscatter measurement techniques that work in spite of the low bandwidth and dynamic range of WiFi radios, new algorithms that separate out the moving backscatter from the clutter that static reflectors produce and then trace the original motion that produced the backscatter in spite of the fact that it could have undergone multiple reflections. We prototype WiDeo using off-the-shelf software radios and show that it accurately traces motion even when there are multiple independent human motions occurring concurrently (up to 5) with a median error in the traced path of less than 7cm.

1 Introduction

Fine-grained human motion tracing, i.e. the ability to trace the trajectory of a moving human hand or leg or even the whole body, is a general capability that is useful in a wide variety of applications. For example, it can be used for gesture recognition and virtual touchscreens (e.g. Kinect style natural user interfaces), activity recognition (e.g. controlling the Nest thermostat depending on intensity of human activity), monitoring of young infants and the elderly, or security applications such as intruder detection. Motivated by these applications, the computer vision community has developed a number of depth sensing based systems (e.g Kinect) to implement motion tracing capabilities in cameras. However these devices are limited because they have a constrained field of view (around 2-4m range with a 60 degree aperture), and do not work in non line-of-sight scenarios, preventing their use in many applications such as whole home activity recognition, security and elderly care.

To tackle these limitations, recent work namely RF-IDraw [43] - has built a motion tracing system using wireless signals. The idea is that users would wear RFID tags, and the motion tracing system would generate transmissions and then listen to reflections of wireless signals from these tags. RF-IDraw then infers the underlying hand motion from changes in reflection signal parameters such as angle of arrival over time. RF-IDraw demonstrates good accuracy and since it uses lower frequencies than light (the 900MHz RFID band whereas visible light is at 600THz), it works in non line-of-sight (NLOS) scenarios and in the dark. However, RF-IDraw has two limitations that restrict its deployability. First, RF-IDraw requires the user whose motion is being traced to wear a special RFID tag on her hands. However, users are accustomed to motion tracing using systems such as Kinect that do not require the user to have any special hardware on them, and changing user habits can be hard. Second, the tracing system requires large antenna arrays of eight antennas with a separation of 8λ, that in their current implementations translates to an antenna array distance of nearly 2.62m. Expecting users in homes to deploy antenna arrays that might span almost an entire room is a big hurdle.

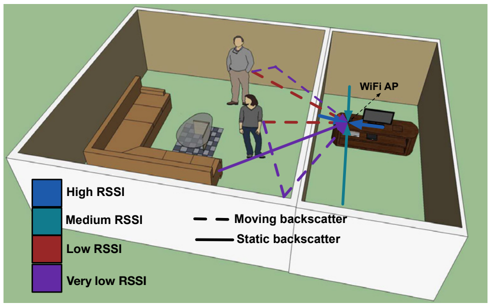


Figure 1: *WiDeo in operation: The compact WiFi AP in the study integrates WiDeo's motion tracing functionality, and can reconstruct the hand movement made by humans in the living room. WiDeo traces motion even though the AP is separated by a wall and does not have a LOS path to the humans, and doesn't require that the humans have any RF devices on them.*

Fig. 1 depicts our goal which is to design a device free, compact motion tracing system. By device free we mean that the humans whose motion is being traced do not need to have any devices on them, whether it's RFID tags or phones. By compact we mean that the motion tracing is implemented on standard WiFi or LTE APs (albeit with minor modifications in hardware and software) and the APs have antenna arrays that they would have had as standard APs anyways. Thus the system is as compact as an AP that is already being deployed. Finally, we would like the system to be non-intrusive, it should be integrated into WiFi and LTE APs that people anyway deploy in their homes and reuse existing packet transmissions for fine-grained motion tracing.

The above requirements pose unique challenges. First, since the system needs to be device-free, it can only rely on natural reflections of the transmitted signals that human limbs naturally produce. These are relatively weak compared to the ones from RFID tags that RF-IDraw uses, and reflections from different objects in the environment cannot be easily distinguished since they are all slightly distorted copies of the same transmitted signal (each RFID tag has its own unique IDs which allows RF-IDraw to distinguish different moving hands because the tags will be different). Second the fact that the system uses a compact antenna array with at most four antennas and regular spacing of $\lambda/2$ makes achieving high spatial accuracy difficult. As the RF-IDraw paper notes, regularly spaced, compact antenna arrays struggle to resolve the spatial angles of incoming signal reflections.

We present WiDeo, a device-free, compact motion tracing system with standard AP antenna arrays. WiDeo only needs 4 antennas per AP, with a spacing of $\lambda/2$ which translates to an antenna array length of 18cm for WiDeo-integrated WiFi APs. At a high level, WiDeo uses the AP's transmitted signals itself as a flash to light up the scene, and then analyzes the *natural reflections* of these transmitted WiFi communication signals from the environment that arrive back at the AP over time to trace any motion that's occurring. WiDeo accomplishes motion tracing through three main components which operate in sequence:

Backscatter Sensor: The sensor analyzes the composite reflected signal received at the WiDeo AP (referred as backscatter) to tease apart the individual reflections coming from each significant reflector in the environment, and calculates each reflection's amplitude, time of flight (ToF) and angle of arrival (AoA). Our key contribution here is a novel algorithm that accurately estimates these backscatter components in spite of the constraints that the humans are device-free, and the limited spatial resolution of the compact antenna arrays. Our key insight is to exploit the natural sparsity that exists in indoor environments; as several empirical studies on indoor MIMO [16, 19] have shown, the number of significant reflectors in an environment is fairly small. WiDeo exploits this insight to accurately measure the backscatter parameters using sparsity aware optimization algorithm.

Second, WiDeo must tolerate limited dynamic range, which causes strong reflections to swamp weak ones, and limited sampling bandwidth, which hides reflections spaced closely in time. Typical WiFi sampling of 80Msps implies a resolution of 12.5ns, or about 6 feet. Our novel algorithms separate weak and closely-spaced reflections despite the limitations of commodity radios.

Declutterer: Reflectors abound in indoor environments, and most of them will be static. The declutterer analyzes the raw set of reflection parameters estimated by the backscatter sensor, and clusters them into groups that correspond to reflections from static and moving reflectors. Further it also eliminates the static reflectors since they are not useful for motion tracing and enables WiDeo to specifically focus on reflections arising from moving objects.

Motion Tracing: This component of WiDeo analyzes the reflections arising from moving objects to predict the underlying motion that could have produced those sequence of reflections and their parameters. We design a novel statistical and sequential estimation framework that predicts the motion that might have taken place, then estimates the changes in reflection parameters the predicted motion would have produced, and compares it with the actual estimated reflection parameters from the backscatter sensor to continuously refine WiDeo's estimate of the motion that occurred.

We design and implement a prototype of WiDeo using WARP radios and simulation environment. The radios are running a standard WiFi OFDM PHY using up to 40MHz, and use 4 antennas with a spacing of 6cm for an overall length of 18cm. We conduct experiments in indoor environments to demonstrate the accuracy of WiDeo's motion tracing. We show that WiDeo can accurately trace multiple sets of fine-grained motion with a median tracing error of less than 7cm, which is comparable to RF-IDraw's performance of tracing error of 5.5cm. Further, the motion tracing has very high resolution, WiDeo achieves the same accuracy even when the multiple humans performing the motion are as close as 2 feet away from each other, which to the best of our knowledge, no prior RF based motion tracing system has demonstrated.

2 Related Work

Fine-grained motion tracing: Vision based systems such as [51, 4] make use of depth sensors (e.g. Kinect) and infrared cameras (e.g. Wii) to trace the fine-grained motion of a user and enable applications such as gesture recognition, virtual touch screens etc. WiDeo, on the other hand, unlike solutions based on depth imaging or infrared, does not require line of sight to work.

RF based systems like [43] and sensor based systems like [23, 26] perform accurate motion tracing but require instrumentation of users. However, WiDeo achieves accurate fine-grained motion tracing in a device-free manner.

RF based coarse motion tracking and gesture recognition: Recent work such as WiTrack and others [34, 7, 6, 5] has shown the ability to *coarsely track full body motion* (not fine-grained motion of human limbs) using radio waves. Other approaches like [35, 24, 33, 30, 49, 45] track human motion by using ultra-wide band (UWB) signals. All of these approaches are also device-free, but unlike these systems, WiDeo is the first device-free fine-grained motion tracing system that can accurately reconstruct the detailed trajectory of a user's free-form writing or gesturing in the air, where the motion may only span a few tens of centimeters. Such free-form tracing capability is not supported by prior work in RF based gesture recognition or motion tracking. For example, [34] presents a state-of-the-art WiFi based interface, yet it only supports the detection and classification of a predefined set of nine gestures. Moreover, many of these systems [6, 5, 35, 24, 33, 30, 49, 45] require GHz of bandwidth unlike WiDeo which works with regular WiFi bandwidths.

There have been approaches like [48, 50, 27, 36] which use existing WiFi infrastructure, with no hardware modifications to achieve device-free human localization and coarse motion tracking, they use coarse information about the environment in terms of Received Signal Strength Indicators reported by WiFi NICs and require extensive war-driving. In contrast, WiDeo requires minor changes to existing WiFi/LTE APs, re-uses the spectrum allocated for communication by performing fine-grained motion tracing using reflections of communication signals that would have been sent for data communication anyway.

Motion clustering techniques: WiDeo also builds on theoretical work on motion segmentation, clustering and classification [41]. These works are targeted at vision applications that use visible light, and deal with taking a collection of pixels that represent the motion and understanding the underlying motion that occurred. WiDeo on the other hand has to deal with RF signal reflections which pose unique challenges such as multiple reflections, noisier measurements and compact, limited sensors (antenna arrays).

Indoor Localization: A large body of work, ranging from classic RSSI based techniques [15, 9, 47, 37] to recent antenna array based techniques [46, 25, 20] exploit already available WiFi infrastructure to provide indoor localization services for radios. They achieve impressive localization accuracy of a few decimeters. Another line of approaches uses single moving antenna to simulate an antenna array [29]. However WiDeo differs from all of them in two fundamental respects. First, it precisely traces fine-grained motion, rather than a static location. Second, its device-free, the traced object does not need to have any RF transmitters on them.

3 Design

WiDeo's goal is to achieve accurate device-free motion tracing of moving objects. To realize this, WiDeo, like standard ToF camera, incorporates four main components:

Flash: This is the light source used to light up the scene; in WiDeo, this is simply the transmission that the AP in which WiDeo is housed is sending for standard communication. In other words, wireless transmissions used for communicating packets act as the flash for the WiDeo.

Backscatter Sensor: This component looks at the backscatter arising from the environment when the AP's transmission gets reflected and arrives back at the AP. The sensor teases out the individual signals emanating from each reflector in the environment as well as estimates each reflection's intensity, angle of arrival and relative time of arrival. The corresponding component in a standard camera are the image sensors which capture the light (aka the backscatter) from objects in the scene and form a picture of the scene.

Declutterer: The captured backscatter contains a lot of reflections from static objects which act as clutter to the reflections originating from the moving object WiDeo wishes to trace. The declutterer component figures out which of the reflections are from objects WiDeo doesn't care about and eliminates them so that motion tracing can focus only on reflections from moving objects.

Motion Tracer: This component looks at the reflections coming from moving objects over time to predict the actual physical motion that could have produced that sequence of reflections.

We omit the description of the flash component since that is a standard AP transmitter. We describe each of the other three components in detail next. For now assume that the AP's receiver can listen to all the reflections from the environment even though it is transmitting at the same time; we describe how we leverage recent work on full duplex to tackle that challenge in § 3.2.2.

3.1 Backscatter Sensor

The sensor's main challenge is to estimate the parameters of each reflection that makes up the received signal. The reflected signals from these reflectors arrive at the AP with different times of flight (ToF), amplitudes and angles of arrival (AoA), but the receiver only obtains the sum of the signals. Let's assume L reflectors are present and that each reflector k applies an unique unknown distortion function $f_k(x(t))$ to the transmitted signal $x(t)$. So the overall backscatter signal $y(t)$ that is arriving back at the AP can be simply written as:

$$y(t) = \sum_{k=1}^{L} f_k(x(t)) \qquad (1)$$

The backscatter sensor's goal is to estimate these functions f_k and then calculate the ToF, amplitude and AoA of the signals reflected from each of these reflectors. As written, the above equation 1 appears intractable, all we know is the transmitted signal $x(t)$ and the overall received signal $y(t)$. How might we tease out the individual reflections? WiDeo makes two novel observations to solve the above under-constrained problem:

Reflector Sparsity: First, WiDeo posits based on recent empirical evidence [16, 19] that the number of significant reflectors in an indoor environment are limited. While there may be many objects, the ones that actually produce sufficiently strong reflections to be visible in the 40 dB of effective dynamic range, which is typical in WiFi radios, are not so numerous. This phenomenon has been extensively proven in empirical wireless communication studies that study the performance of MIMO which critically depends on the number of independent reflectors in an environment [16, 19]. In WiDeo's case, this means that the number of reflectors that could have contributed significantly to the overall signal is limited.

Narrowband Transformations: The second key observation is that WiDeo uses narrowband communication signals and radios as the flash/light source. By narrowband we mean that the signals generated or received by the WiDeo device (the AP) are filtered to only let the signal within the bandwidth, which conforms to FCC regulation, used for communication to come through. For example, if we are using the WiFi channel of width 40MHz at center frequency 2.42GHz, then a passband filter of width 40MHz centered at 2.42GHz is applied at the transmitter and the receiver. Filtering by a passband filter can be modeled as convolution with a sinc pulse of the same bandwidth in the time domain [1]. So the reflected signal (after including the attenuation and delay) is now convolved with a sinc pulse. So the signal that arrives back from a single reflector is actually given by:

$$f_k(x(t)) = \big(\alpha_k x(t-\tau_k)\big) \otimes \mathrm{sinc}(Bt) \qquad (2)$$

where B is the communication bandwidth of the signal, α_k is the complex amplitude and τ_k is the overall delay of the reflection or the Time of Flight (ToF) for the k^{th} reflector, and $\otimes$ represents the convolution operator [39]. Eq. 2 can equivalently be written as:

$$f_k(x(t)) = \big(\alpha_k \mathrm{sinc}(B\times(t-\tau_k))\big) \otimes x(t) \qquad (3)$$

If there are L reflectors, then all L reflections will undergo different attenuations and ToFs, add up over the air and then get convolved with a sinc pulse. Therefore the overall signal is given by:

$$y(t) = \big(\textstyle\sum_{k=1}^{L} \alpha_k \mathrm{sinc}(B\times(t-\tau_k))\big) \otimes x(t). \qquad (4)$$

The sensors now first calculate the overall transformation $h(t)$ that has happened to the transmitted signal $x(t)$, i.e. $y(t) = h(t) \otimes x(t)$ where $h(t)$ is essentially the sum of the transformations applied by all the reflectors. This is classic channel estimation that's used in standard communications (after all every receiver estimates the channel that has transformed the transmitted signal to be able to decode). We refer the reader to the following literature [8] for a review of the different techniques that can be used.

However, WiDeo's problem is quite harder than standard channel estimation which only cares about the overall transformation. Although, WiDeo knows the overall channel $h(t)$, it needs to figure out the amplitudes and time shifts of the sinc pulses that are summed up to produce the overall channel $h(t)$. The equation that WiDeo has to solve is therefore given by:

$$h(t) = \textstyle\sum_{k=1}^{L} \alpha_k \mathrm{sinc}(B\times(t-\tau_k)) \qquad (5)$$

We can rewrite the equivalent equation in the digital domain (after all WiDeo will be working in the baseband domain after ADC sampling) as:

$$h[n] = \textstyle\sum_{k=1}^{L} \alpha_k \mathrm{sinc}(B\times(nT_s-\tau_k)), \qquad (6)$$

where T_s is the sampling time of ADC. WiDeo's goal is to solve the above equation to determine α_k and τ_k for all reflections.

To tackle this, we now exploit the sparsity observation that the number of significant reflectors in an environment is limited to a handful (typically on the order of 10-15). Specifically, we attempt to find the smallest number (less than 20 in our implementation) of scaled and shifted sinc pulses that could have summed up to produce the overall channel response. Mathematically, we are attempting to solve the following problem:

$$\begin{aligned} \min \quad & \textstyle\sum_n (h[n] - \sum_k \alpha_k \mathrm{sinc}(B\times(nT_s-\tau_k)))^2 + \lambda_r|\alpha|_0 \\ \text{s. t.} \quad & \tau_k \geq 0, |\alpha_k| \leq 1, k = 1:L, n = -N:N. \end{aligned} \qquad (7)$$

Note that the above problem is similar to classic problems in compressive sensing [17, 42, 14]. Like in compressive sensing problem, we are trying to find the minimum number of non-zero components (each component corresponds to a reflector) and the corresponding scaling and shifting factors that best explain the observed channel $h[n]$. The sparsity of the number of reflectors is coerced by the term $\lambda_r|\alpha|_0$, where λ_r is a positive regularization term and $|.|_0$ is the number of non-zero terms in the amplitude vector. However there is one major difference, WiDeo's problem is trying to find the best sparsest combination of parameterized *continuous* basis functions (the sinc pulses parameterized by continuous shift factors), whereas classic compressive sensing is finding the sparsest combination of discrete finite sized vectors that produces some overall vector. We omit the mathematical details here for brevity, but refer the reader to a large body of literature on solving these sparse estimation problems [18, 40, 31]. WiDeo's contribution is to show that the backscatter sensing problem can be formulated using sparsity and compressive sensing intuition.

3.1.1 What if the reflectors are closely spaced?

The above description didn't make any mention of how closely spaced the reflectors are. For example, if the two reflectors are a foot apart, their reflections will arrive at the AP within two nanosecond of each other (wireless signals travel a foot per nanosecond and reflection for objects a foot apart takes 2 nanoseconds). But sampling rates of wireless communication radios are at best around 100Msps (Mega samples per second), which means that two samples are spaced 10ns apart. How could then WiDeo estimate the parameters of the two reflectors that are closely spaced to an order of magnitude closer in time than the sampling period? Even if two reflections are closely spaced in time because their reflectors are almost at the same distance from the AP, they are likely to be at different spatial angles (otherwise they would be the same reflector!). So the spatial dimension provides us the ability to separate reflections in space when they are close in time. The heuristic works in the other direction too, if two reflectors are at the same AoA (because they are on the same radial line), they are likely at different delays and can be separated out.

How do we use this insight to separate out reflections? The intuition is that if the WiDeo AP has an antenna array (typical APs have 4 antennas), then the specific AoA of each reflection imposes a constraint on how the phase of that reflection changes across space. Specifically if the antennas are laid out equidistant at distance d in a straight line, the so called uniform linear array (ULA), and if the AoA is θ, then the relative phase between the signal at any two consecutive antennas is given by $(\phi(\theta) = 2\pi d \sin(\theta) c/\lambda)$, where c is the speed of light in air and λ is the wavelength of the RF carrier. Assuming that there are four antennas in the WiDeo's AP we call the following vector $[0, \phi(\theta), 2\phi(\theta), 3\phi(\theta)]$ of phase differences of all the antennas with respect to the first antenna as the relative phase constraint vector.

In general when more than two backscatter signals are present, each of these backscatter signals arrives at all four antennas, but based on the AoA of these signals the relative phase constraint vectors of these signals will be different. WiDeo uses this insight in the following way. In addition to finding the best sparse signals as described by 7, WiDeo imposes an additional constraint that these estimated sparse solutions should strictly follow the phase vector constraint imposed by the ULA structure leading to the following problem for Ψ antennas:

$$\begin{aligned} \min \quad & \sum_m \sum_n \Big(h_m[n] - \sum_k \alpha_k e^{-i(m-1)\phi(\theta_k)} \operatorname{sinc}(B \times (nT_s - \tau_k))\Big)^2 \\ & + \lambda_r |\alpha|_0 \\ \text{s. t.} \quad & \tau_k \geq 0, |\alpha_k| \leq 1, k = 1:L, n = -N:N, m = 1:\Psi. \end{aligned}$$

The $e^{-i(m-1)\phi(\theta_k)}$ term in the optimization objective function is encoding the phase constraint that arises from a specific AoA. In essence, while many signals can fit the time domain constraint given by 7, only few of them can satisfy the relative phase constraint vector thereby further limiting our solution space and hence increasing the accuracy of our estimates despite the closeness of these signals in time. The matching relative phase constraint vector of ULA has one-to-one relationship with AoA, thus using this process we can simultaneously estimate the AoA of the backscatter signals in addition to their amplitude and ToF.

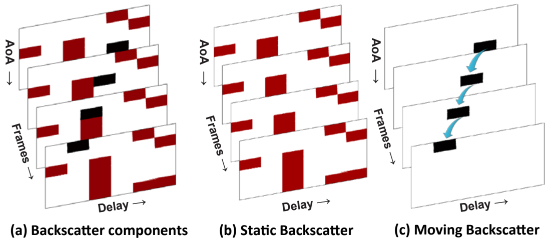
To summarize using the above technique, the sensor outputs a set of reflections with their associated three tuple of parameters: amplitude, ToF and AoA. The next step is eliminating the numerous reflections from static objects that act as clutter to motion tracing problem which we describe below.

3.2 Declutterer

Reflectors abound in the environment and their reflections end up cluttering the backscatter, making it hard for WiDeo to focus on the reflections arriving from the moving object that's being traced. Tracing accuracy can be greatly improved if this clutter can be eliminated. There are two kinds of clutter in decreasing order of harmfulness. The first are reflections from objects nearby whose relative strength w.r.t. to the moving object reflection is greater than the dynamic range of the WiDeo receiver. In this case, the reflection from the moving object is completely lost in the quantization noise and motion cannot be traced. The second is clutter whose strength is within the dynamic range relative to the moving object's reflection. Here information is not lost, but it becomes harder for the tracing algorithm to recover the original motion. WiDeo's declutterer handles both kinds of clutter and eliminates them. We start by describing how to handle the second kind of clutter by guessing which reflections are from moving objects, and then describe how to eliminate the rest of the clutter including the nearby reflectors.

3.2.1 Eliminate Reflections of Static Objects

WiDeo uses a heuristic to loosely identify reflections that are likely to have come from moving objects. The basic idea is to look across sequences of backscatter sensor measurements as shown in Fig. 2, and then make an association of which reflections haven't changed in value and which have. The idea is that the reflections that have continuously changed their parameters (their amplitude, AoA and ToF) will include reflections from moving objects. Everything else is classified as static clutter that has to be eliminated.



(a) Backscatter components (b) Static Backscatter (c) Moving Backscatter

Figure 2: *The figure represents backscatter components obtained from a simulated hand movement in a typical indoor scenario using ray tracing software [2]. The backscatter components collected in each time interval are presented as an image snapshot. The horizontal and vertical axes correspond to ToF and AoA respectively. Each colored pixel corresponds to a backscatter component. Different snapshots stacked one over the other correspond to set of backscatter components obtained in consecutive time intervals. The majority of backscatter components are contributed by static environment, which are shown in the same color to provide contrast with moving backscatter.*

The key question then is to look at snapshots of backscatter over time, associate the backscatter parameters that we believe are coming from the same reflector and then apply the above heuristic . Each snapshot is made up of as many backscatter points as number of reflections, and each point is associated with a three tuple of amplitude, ToF and AoA. WiDeo keeps track of a moving window of such backscatter snapshots (in our current implementation the last 10 snapshots are maintained). The first step is to associate points which are generated from the same reflector between every two successive snapshots, even if the reflector moved between those two snapshots. To do so, we invent a novel point association algorithm across snapshots based on minimizing the amount of change between consecutive snapshots.

Identification of Static Reflections: The algorithm starts by calculating the pairwise distance between every pair of backscatter points in successive snapshots. Distance is defined as the absolute difference in the three parameters (amplitude, ToF and AoA) squared and summed after appropriate normalization. Note that this metric is calculated for all pairs of points, so there would be n^2 distances where n is the number of backscatter points in a snapshot. The goal is to figure out the specific pairings where points in each pair of snapshots are generated by the same backscatter reflector.

Our key insight is that for static objects, the points corresponding to backscatter reflections from that static object in successive snapshots should be at zero distance with respect to each other because by definition they did not move and the associated parameters did not change. Further even for the points that correspond to moving reflectors, given how slow human motion is relative to the length of a backscatter snapshot (a millisecond), the distance between points in successive snapshots that correspond to the moving object is small. So if we can pair the points up such that the overall sum of the distance metrics for these paired points is the minimum among all possible pairings, then very likely we will have associated the right sets of points together.

How do we determine the right point association between successive snapshots? This is a combinatorial assignment problem where we first pass distances between all pairs of points as the input and then pick the set of pairs that minimize the overall sum of distance metric among them. A naive algorithm would be to enumerate all possible assignment of point pairs, which would require evaluation of n! assignments for snapshots with n points. To reduce the complexity, we turn to a classic algorithm in combinatorial optimization known as the Hungarian algorithm [28] which runs in polynomial time. We omit a full description of the algorithm for brevity however, the algorithm is best visualized in terms of a bipartite graph $G = (F1, F2, E)$, where points from the first snapshot are vertices in the set $(F1)$ and points from the second snapshot are in the set $(F2)$ and the edge set (E) consists of all possible edges between vertices in the two sets. The weight of each edge is the distance metric between the backscatter parameters corresponding to the

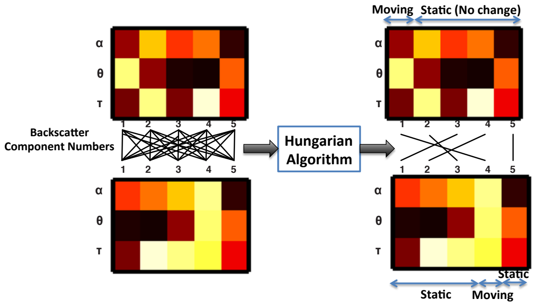


Figure 3: *This figure illustrates application of Hungarian algorithm for a subset of backscatter components obtained in the experiment narrated in Fig. 2. The left side represents the backscatter components in two successive snapshots. The color of each pixel is a representation of the value of α(power in dBm), θ(in degrees), or τ(in ns) according to the appropriate row. We design a distance metric between each component in the first (top) snapshot and each component in second (bottom). The distance thus obtained are represented as edges with appropriate weights (not shown in the figure for clarity). We want to find the matching with minimum weight in the above bipartite graph. Applying Hungarian algorithm results in the least weight matching presented on the right, thus providing a way to associate backscatter components in the two snapshots.*

points the edge connects. The goal of our algorithm is to find a matching with minimum cost, as shown in Fig. 3.

Eliminating Static Clutter: The next step is to eliminate the clutter caused by static backscatter reflectors. This step immediately follows from the above computation; we identify the pairs of points whose distance metric is close to zero, and stays so for at least a fixed number of snapshots (typically around 10 snapshots corresponding to those reflectors being static for 10ms). When we find such points, we declare them to be part of the static clutter. These points are then eliminated from the snapshots and the only points left are those that the algorithm believes to be coming from moving reflectors.

The static clutter elimination step also naturally provides the detection of a new motion that is starting. For example, let's say we start with a completely static environment; in the steady state the declutterer block won't report any parameters because eventually all of the components will be declared static and eliminated as clutter. When a new motion starts and generates new backscatter components, the sensor will report these parameters to the declutterer which will classify them as moving points. Such points are grouped together and passed to the motion tracing block, described in section 3.3, as a new moving object that needs to be traced.

3.2.2 Eliminating Clutter from Nearby Reflectors

In many scenarios, we may have a nearby reflector that is producing strong reflections. If these reflections are stronger than the reflections from the moving object that WiDeo wishes to trace by more than the dynamic range of the radio, all information about the moving object will be lost in the quantization error of the ADC at the receiver. Further remember that WiDeo aims to listen to reflections from the environment while the WiDeo AP is transmitting signals for communication. The transmitted signal also directly leaks through to the receiver and causes interference.

WiDeo's observation is that such clutter is essentially a form of self-interference, and recent work on full duplex radios can be used to eliminate such clutter [12]. Full duplex radios have to solve a similar problem, they have to cancel their own transmitted signal's leakage and reflections that arrive back at the receiver. This self-interference also incorporates reflections from the environment, and recent work has developed sophisticated interference cancellation techniques that can eliminate the self-interference to the noise floor [12]. WiDeo leverages this work. We provide a brief description below, but refer the readers to [10] for a detailed description. WiDeo's contribution is showing how full duplex can be used to build imaging applications rather than the communication applications that full duplex research has focused on.

Conceptually, full duplex radios consist of a programmable canceler component that consists of both analog and digital cancelers. The canceler's main component is a programmable filter which attempts to model the distortions that the transmitted signal goes through before arriving back at the same radio's receiver as self-interference. The canceler takes the transmitted signal as input, passes it through the programmable filter, and then subtracts the filtered signal from the received signal to completely eliminate self-interference.

Note that in traditional full duplex radios, the goal is to completely eliminate the self-interference. WiDeo however is different, in fact some of the self-interference may be coming from moving objects that we do not want to cancel since we want to infer the motion from them. So WiDeo implements a novel modification to traditional full duplex self-interference cancellation. It uses the backscatter sensor measurements to program the filter to only model the *static and strong* reflectors that act as clutter, but intentionally leaves the components that would also have modeled the moving reflectors out. WiDeo figures out which backscatter components correspond to moving reflectors using the static clutter detection algorithm described in the previous section. Thus cancellation is selectively applied only to the static and strong clutter components. Specifically the programmable canceler filter is tuned to implement the following response:

$$h_{c_m} = \sum_{k=1}^{L'} \alpha_k e^{-i(m-1)\phi(\theta_k)} \mathrm{sinc}(B \times (t - \tau_k)) \qquad (8)$$

where $\alpha_k, \tau_k, \theta_k$ is the amplitude, ToF and AoA parameters for all L' unwanted reflectors, and h_{c_m} is the response of cancellation filter attached to the mth antenna.

This completes the design of the declutterer component. At this point we have a set of snapshots with points that correspond to moving objects. Further points in successive snapshots are associated with each other if they belong to the same moving reflector. However note that this does not mean we have traced the original moving object itself, all we have isolated is the multiple backscatter reflections from it. The next step is to trace the original object and its motion which produced the snapshots with the moving points.

3.3 Tracing the Actual Motion

Each WiDeo AP sends the isolated backscatter measurements arising from moving objects it computes from the previous step to the central server. Whenever a new motion starts, its quite likely that many of the WiDeo APs will detect the backscatter measurements from this new motion. The server collects backscatter snapshots over a period of 10ms from all participating radios, and assumes that any moving backscatter detected by any of the radios are coming from the same object. The heuristic implicitly makes the assumption that two new and independent human motions won't start within an interval of 10ms. Given the timescales at which human motion happens, 10ms is a negligible amount of time and we believe that such asynchrony is very likely in practice. Note that this does not mean that two independent motions cannot be occurring simultaneously, we are only making the assumption that they don't start within 10ms of each other.

3.3.1 Localizing the origin of the motion

The first step the server implements is actually to localize the origin of the motion that just started. The server has measurements from multiple radios across multiple snapshots, and very likely the new motion will be detected at many of these radios. So, how might we estimate the location of the new motion? The idea is that the measurements collected at the WiDeo APs imposes constraints on where the moving reflector is located. We demonstrate the idea using the AoA measurement. Let's say the locations of the M WiDeo APs involved in motion tracing are given by $(x_i, y_i), i = 1, \ldots, M$. Similar to many other state-of-the-art localization systems [46, 38] using WiFi, the locations of the APs (or the anchors) are assumed to be known in advance. Let the AoA measurements of the reflector at the APs be denoted by $\theta_i, i = 1, \ldots, M$ and the current estimate of the object's location is (x, y). So the most likely location of the object is one that minimizes the following metric:

$$\begin{array}{ll} \min & \sum_{i=1}^{M} (\bar{\theta}_i + b_{\theta_i} - \theta_i)^2 \\ \text{s. t.} & \bar{\theta}_i = \text{AoAULA}((x, y), (x_i, y_i)) \end{array} \tag{9}$$

The above equation is stating the fact that the predicted angle of arrival at each of the WiDeo APs given the estimated location of the reflector and the location of the APs must closely match the actual AoA measured by each of the APs. The function AoAULA$((x, y), (x_i, y_i))$ computes the AoA seen by the tracing radio located at (x_i, y_i) from a reflector located at (x, y). However there is a new factor b_{θ_i} that represents the bias to model multipath reflections. This is because the moving backscatter not only corresponds to the direct backscatter from the object but also the backscatter from the reflections of the backscatter. For example, if a backscatter reflection from a moving object is further reflected by a wall before arriving at the AP, the ToF parameter will have a constant bias that reflects the extra time it takes to traverse the extra distance corresponding to going to the wall and reflecting off it. Similar bias exists for both the amplitude and AoA measurements. Further the bias values are unknown and hence are a variable in the optimization. The value of (x, y) that minimizes the above metric is likely the best estimate of the location of the reflector.

We can also use other parameters like ToF and power to estimate the location of the target. In our actual implementation, we solve a more sophisticated optimization problem than the simple optimization problem in 9. Specifically, WiDeo uses AoA, ToF, and backscatter signal strength measurements over multiple frames for the particular backscatter, say J frames, and declares the origin of the motion as the location that minimizes the following objective function described by Eq. 10

$$\sum_{j=1}^{J} \sum_{i=1}^{M} [(\bar{\alpha}_i - \alpha_{ij})^2 + (\bar{\tau}_i + b_{\tau_i} - \tau_{ij})^2 + (\bar{\theta}_i + b_{\theta_i} - \theta_{ij})^2], \tag{10}$$

where α_{ij}, τ_{ij}, and θ_{ij} are the power, ToF, and AoA respectively of the backscatter observed by the i^{th} AP in the j^{th} frame and the variables $\bar{\alpha}_i$, $\bar{\tau}_i$, and $\bar{\theta}_i$ are the values of respective backscatter parameters that would have been observed at the APs if the object was actually located at that particular location. We follow a simple path loss model [21] to describe the relation between the location of the object and the backscatter signal strength $\bar{\alpha}_i$. The variables b_{τ_i} and b_{θ_i} represent the bias in ToF and AoA respectively due to reflections of the backscatter from the object. This problem of minimizing Eq. 10 is non-convex, therefore we apply a widely used heuristic known as sequential convex optimization to solve it [13].

We note that Eq. 9 as such is an ill-posed problem without a unique solution because each AP introduces its own bias terms for backscatter parameters. However, in Eq. 10, by collecting measurements over enough number of backscatter frames, the number of measurements become greater than the number of variables and the optimization problem becomes well-posed. Further the parameters of simple path loss model used to model $\bar{\alpha}_i$ are also estimated as part of the minimizing Eq. 10 and need not be known ahead of time.

3.3.2 Tracing Motion

Once the newly detected moving object is localized, the next step is to trace the object's motion as it moves and produces new measurements via our backscatter sensor. Remember that the new measurements are naturally associated with the measurements from the previous snapshots via the declutterer described in § 3.2. So the algorithm has already clustered backscatter measurements coming from the same moving reflector together, and we can operate the motion tracing algorithm on each cluster of measurements separately. Hence we describe the tracing algorithm as if there is a single motion occurring and a single set of backscatter measurements being produced from it across successive snapshots.

Our approach to this problem is to build a dynamical model about the motion that is occurring and progressively refine its parameters. There are several parameters to the motion model: current position of the object, velocity, direction of motion, acceleration, and bias in each backscatter parameter due to the indirect reflections. Both the bias and initial position variables are initialized using the output of the localization algorithm in the previous step. Velocity, acceleration and direction of motion are initially set to zero and then updated over time as new measurements come along. Note that we also allow the bias parameter to change over time, after all as the object moves, the bias for each parameter changes.

The key insight is as follows: at every point in the traced motion, given the estimate of the motion model at that instant, WiDeo can predict what the backscatter sensor measurements for that moving reflector should be (given the estimate for the locations of the reflector and the WiDeo AP and the biases in the parameters we can calculate the expected amplitude, ToF and AoA of the reflections). WiDeo also of course has access to the actual backscatter sensor measurements at that instant for the same moving reflector, so we can calculate the error between the predicted and the actual backscatter measurement. The goal of the motion tracing component is to *minimize the sum of these backscatter prediction errors over the entire motion trajectory in a sequential fashion*. The algorithm proceeds in three steps at each time instant:

Model based prediction: In this step, WiDeo calculates the new position of the reflector given the previous position and motion model parameters namely, velocity and acceleration. It then uses this extrapolated position along with the estimates of the bias for the backscatter parameters to calculate what the new values of the amplitude, ToF and AoA of the reflection should be.

Backscatter prediction error computation: Compute the difference between the above predicted and measured backscatter parameters.

Model update from error: Update motion model parameters such that the overall backscatter prediction error is minimized across the entire trajectory. The update step uses a classic technique in dynamic estimation: the Kalman filter [44]. Kalman filter theory shows that assuming the measurement noise and motion modeling error is Gaussian, the update is dependent on two factors. First factor is the size of the prediction error itself, i.e. if the error is large then a larger update to the model is required and vice-versa. The second factor is a gain term that modulates this error term. The gain factor is chosen such that the accumulated error between all the observations of the measured backscatter parameters so far and the best prediction that the motion model can make is minimized. In essence the gain signifies the effect of accumulated errors in the motion model, for example if the measurements are noisy the gain should be chosen small to account for the unreliable nature of the measurement and vice-versa. We omit the proof and refer the readers to [44] for a more detailed mathematical treatment of the Kalman filter and how to compute the gain factor given the motion model and history of backscatter measurements and prediction errors.

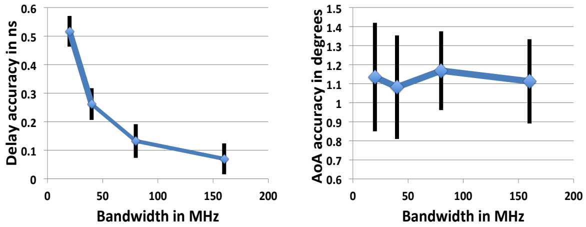


Figure 4: *Accuracy of WiDeo's algorithms in estimating the delays and AoAs of backscatter. WiDeo achieves an accuracy of 300ps and* 1.2 *degrees (with error bar representing standard deviation) at 40MHz bandwidth used in WiFi signals.*

The convergence of the motion tracer takes a few snapshots, after this point it constantly updates its motion model parameters. Reconstructing the motion is now akin to starting with the initial point and performing a directional piece-wise integration using the speed and direction of motion parameter at each time step. An instance of the above algorithm is executed for each detected motion.

4 Evaluation

We implement a prototype of WiDeo using the WARP software radios using WiFi compatible OFDM PHY with a bandwidth of 20MHz at 2.4GHz. The radio is set up to use 4 antennas and all RX chains are phase synchronized like in a MIMO radio. The spacing of the antennas is $\lambda/2$ and the overall width of ULA is 18cm. The declutterer is designed using analog cancellation circuit boards based on the design described in [11]. From the time it receives information about the clutters to be canceled, the declutter takes few microseconds to remove their effect and improve the dynamic range. The optimization algorithms that measure the backscatter parameters and the rest of the tracing algorithms are implemented in a host PC in C using the cvxgen toolbox [32] and Matlab. Although the current implementation of WiDeo is not realtime we believe it is possible with a few architectural changes and speed optimizations in the future.

4.1 Back-scatter sensor benchmarks

We start with micro-benchmarks of the backscatter sensor that underpins motion tracing. The goal here is to demonstrate that WiDeo's backscatter measurement algorithms provide high accuracy and fine resolution.

Accuracy: We first measure WiDeo's accuracy in measuring backscatter parameters. Given the complex geometry of indoor environments, a natural question is how do we know ground truth for all the multipaths to evaluate the accuracy of WiDeo? We perform controlled experiments by connecting the RX chains with wires from the transmitted chain. The lengths of the wires are varied to provide different delays, attenuators on each wire provide tunable amplitude, and phase shifters are introduced to simulate AoA. This wired experiment can create 10 different backscatter components. To mimic realistic indoor reflections, we vary the lengths and attenuations by sampling it from an indoor power delay profile [19], and AoA is picked uniformly at random. We vary the receiver bandwidth from 20MHz to 160MHz.

Since WARP radios can only support up to 40MHz bandwidth, we use signal analyzers for the higher bandwidth experiments. Higher receiver bandwidth is expected to help improve accuracy because we are getting finer-grained observations in time due to higher sampling rates. However, the default configuration for WiDeo unless stated otherwise is WARP radios with 20MHz bandwidth.

Fig. 4 plots the overall estimation accuracy for delay and AoA of the backscatter components as a function of bandwidth, we omit amplitude results for brevity (their accuracy was within 1dB). As we can see WiDeo provides extremely high accuracy, measuring delay to within 0.3ns accuracy for a bandwidth of 40MHz, the most commonly used WiFi bandwidth. Further AoA accuracy is 1.2 degrees at 40MHz bandwidth. Accuracy improves slightly for delay estimation with bandwidth, which is expected since we now get more closely spaced samples that helps discern delay better. AoA accuracy is not affected much by bandwidth since that is primarily determined by the number of antennas.

Resolution: Next we conduct an experiment to measure WiDeo's resolution, i.e. how close two backscatter reflectors can be before WiDeo's algorithms fail to disambiguate their respective parameters? First, we create two backscatter components whose delays are far from each other by using wires of different lengths. We then slowly decrease the relative delay and measure at what relative delay the accuracy is a factor of two worse than in Fig. 4. Next we repeat the same experiment, but instead of delay, we make the AoA of two backscatter components very close to each other and check at what relative AoA the accuracy is a factor of two worse than in Fig. 4. The results are presented in Fig. 5.

WiDeo can resolve delay to 2ns, distinguishing two gesturing humans separated by only one foot. WiDeo can resolve angle to 5 degrees, distinguishing humans 1 foot apart at 12 feet away.

Range and Dynamic Range: A third benchmark is how weak a backscatter signal can be before it cannot be estimated by WiDeo. Clearly if a backscatter is weaker than the noise floor of the receiver radio (-90dBm), then WiDeo cannot detect it. But how much above the noise floor does the backscatter have to be for accurate measurement? We repeat the accuracy experiment shown in Fig. 4 by picking the parameters for 9 components from the power delay distribution while progressively decreasing the strength of the 10th backscatter component.

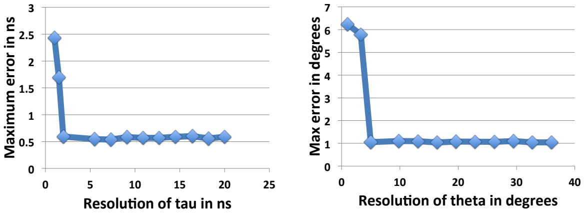


Figure 5: *WiDeo can accurately measure parameters even when the backscatter are spaced only 2ns apart in time or 5 degrees apart in spatial orientation.*

Fig. 6 (on left) plots estimation accuracy of different backscatter parameters as a function of the received strength at the radio. When the backscatter component is weaker than -70dBm (i.e less than 20dB above the noise floor of the receiver), WiDeo's accuracy degrades to around 6ns for the delay. In practice this means that the motion that is being traced needs to happen within 16 feet radius of the radios for high backscatter sensor accuracy. Note that the range of motion tracing can be more than 16 feet as motion tracing may not need parameters to be highly accurate.

Another related benchmark is the resilience of WiDeo in scenarios where there is backscatter from a nearby reflector and the motion we actually want to trace is farther away and producing weak backscatter. To test this we conduct a controlled experiment where there are two backscatter reflectors, one nearby whose strength is kept constant at 10dBm while the other one is made weaker and weaker. We plot the accuracy of backscatter measurement for the weaker component as a function of the difference in strength w.r.t. the strong backscatter component in Fig 6(on right). WiDeo accurately measures components as weak as 80dB below the strong reflector, well beyond the radio's 40dB dynamic range. This works because the declutterer estimates the strong component, then cancels it completely all the way down to the noise floor.

Note that both the maximum range and the dynamic range of WiDeo is limited by the noise floor of the radio being used and the transmitted power by the sensor, and not due to the limitations of WiDeo's algorithm. This is because WiDeo's cancellation can cancel specified reflections all the way to the noise floor. If the cancellation were imperfect and doesn't reach the noise floor; for example, a 20dB residue will limit WiDeo to sensing signals above -50dBm rather than the -70dBm shown in Fig 6,

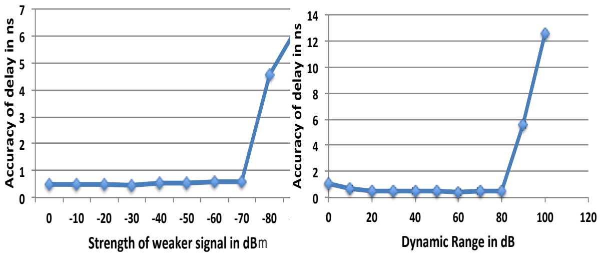


Figure 6: *(on left)WiDeo can accurately estimate backscatter parameters for reflections that are as weak as* -70*dBm. (on right) It can also accurately estimate parameters for very weak backscatter components even when there is a strong backscatter component present which is 80dB stronger.*

which would reduce the range as well to 2 feet.

4.2 Motion tracing benchmarks

We now evaluate WiDeo's ability to accurately trace motion in indoor environments. We calculate two metrics
Location accuracy: This is the accuracy of the localization of a motion that is detected by WiDeo. We use Euclidean distance between the centroid of the ground truth motion and the estimated motion as the metric.
Motion tracing accuracy: This is the accuracy of the traced motion. The metric we used is the root mean square error of the traced motion which we calculate by computing the distance of each point in the traced motion with the ground truth motion trace at that point. The distances are squared and added up and normalized by the number of points before taking the square root. Hence, the metric represents the motion tracing error in meters. Similar to [43], we remove any offset between the ground truth motion and traced motion.

The locations tested for motion tracing accuracy spans all scenarios: non line-of-sight (NLOS) to any of the tracing radios, LOS to a subset of the tracing radios and through walls in an indoor environment spanning 600sq.ft. By default, unless stated otherwise, the number of tracing radios is fixed to three and they are deployed at three fixed but arbitrarily picked locations in the testbed. The motions we trace are actually humans sketching various shapes with their hands. By default, unless stated otherwise, we have two humans performing motion concurrently in our experiments.

We could not find any recent system that implements fine-grained motion tracing within the design requirements of WiDeo: namely being device-free, compact and one that uses existing communication signals and spectrum. RF-IDraw [43] as discussed before is not device-free, nor compact. Other recent work such as WiTrack [6] is device free but implements coarse tracking of the entire human body moving, but cannot track fine-grained motion of human limbs. Hence we refer the reader to § 1,§ 2 for a qualitative comparison to these related systems.

Our experimental results show the following

- WiDeo accurately traces motion, it achieves a median localization accuracy of 0.8m and motion tracing accuracy of 7cm.
- WiDeo can accurately trace multiple independent motions, tracing as many as five independent and concurrent motions with an error less than 12cm.
- WiDeo's resolution is 0.5m, i.e. if the two independent motions are occurring within half a meter or higher of each other, WiDeo can trace them accurately.
- Accuracy improves modestly with the number of radios involved in the tracing. When we increase the number of radios to five, localization accuracy improves to 0.7m, whereas motion tracing accuracy improves to 6cm.

4.2.1 Motion tracing experiments

We use a SPEAG hand [3] to perform motion tracing experiments. This model hand is designed to have same dimensions and absorption/reflection characteristics as that of a typical human hand in 2.4GHz frequency range.

This hand is placed over a chart with figures of different shapes like the one shown in Fig. 7. Several markers are drawn on the shape and the backscatter is captured by WiDeo's APs when the SPEAG hand is placed on each of these markers. The markers are spaced apart by approximately 5cm so as to emulate a scenario where WiDeo collects measurements every 10ms when a human hand is moving at a speed of 5m/s [22]. The ground truth location for each marker on the chart is obtained by using laser range measurements and architectural drawings. In Fig. 7, the shape in the blue shown in the right is found using such laser measurements. By placing the model hand in all the locations of the chart sequentially, we emulate the scenario where a human hand traces the particular trajectory whose ground truth is accurately determined.

We conducted experiments in scenarios with one, two, and all three APs in LOS. WiDeo's accuracy is tabulated in Fig. 8. WiDeo achieves an accuracy of 5.1cm when the APs are in LOS, and is still quite accurate at 12.8cm when two of the APs are in NLOS.

4.2.2 Understanding WiDeo's motion tracing

Because of the time consuming nature of the data collection procedure for the above testbed experiments, we

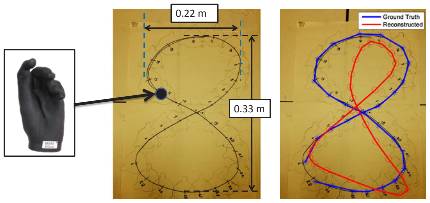


Figure 7: *(Left) A chart with figure of 8 with multiple markings where SPEAG hand (in inset) was placed and the data was captured by WiDeo's AP. (Right) Ground truth data obtained using laser range finder (in blue) along with the motion trace reconstructed by WiDeo (in red) using 3 APs.*

Motion	Localization Accuracy (m)		Motion Tracing Accuracy (cm)	
	Testbed	Wireless InSite	Testbed	Wireless InSite
All AP LOS	.54	.54	5.1	5.3
1AP NLOS	1.1	1.1	8.5	8.4
2AP NLOS	1.61	1.63	12.8	12.5

Figure 8: Median accuracy for different motion shapes obtained using SPEAG hand and Wireless InSite tool.

can only perform a limited number of experiments using it. To extensively test the motion tracing accuracy of WiDeo under more diverse conditions, we simulated the entire system in an electromagnetic emulation environment called Wireless InSite [2]. Wireless InSite is a ray tracing based tool to accurately model RF propagation in any indoor environment with walls and other objects. This tool enables us to emulate complex indoor environments in which WiDeo will be used, as well as know the ground truth for every experiments. To demonstrate Wireless InSite produces similar results, we modeled the testbed described above and then collected data for the same scenarios in Wireless InSite. We emulated the dynamic range and progressive interference cancellation on the data obtained from Wireless InSite simulation. Fig. 8 compares the accuracy of motion tracing achieved with Wireless InSite data with that obtained with the physical experiments. We see that the two results match very closely which is due to Wireless InSite's ability to accurately model indoor RF environments. Hence, in the rest of the sections, we use Wireless InSite to analyze performance of WiDeo in more detail.

4.2.3 WiDeo's motion tracing performance

We now evaluate the WiDeo's motion tracing accuracy by conducting extensive experiments using Wireless InSite. Specifically, we vary the placement of the two moving humans arbitrarily in the testbed across 100 different locations. We calculate the median localization error and the root mean square error of the traced motion. We plot the CDFs in Fig. 9.

WiDeo achieves a median localization error of 0.8m and a median tracing error of 7cm. The tail errors are often cases where the human motion is happening in a dead zone where the backscatter to any of the tracing radios is weaker than -80dBm. In these cases the backscatter measurement itself has worse accuracy which translates to poor accuracy for motion tracing. However WiDeo still achieves a motion tracing accuracy better than 15cm in 90% of the scenarios.

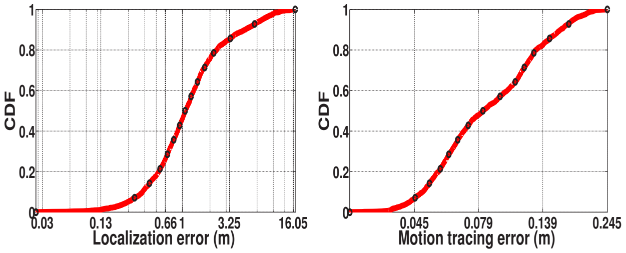


Figure 9: *WiDeo's motion tracing is extremely accurate; it traces fine-grained motion with a median localization error of 0.8m and motion tracing error of 7cm.*

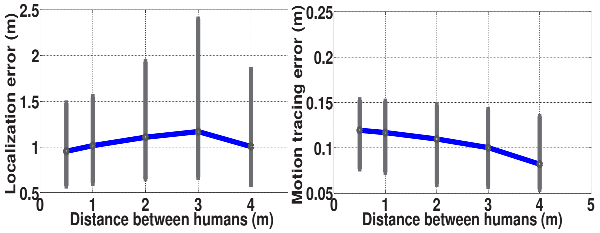


Figure 10: *WiDeo provides high resolution motion tracing, it can accurately trace two independent motions occurring even if they are only spaced 0.5m apart (with error bar representing standard deviation).*

4.2.4 Resolution

Many applications that might build upon WiDeo's motion tracing capability care about resolution, i.e. how close can two independent human motions be occurring and WiDeo can still trace them accurately (e.g multi-player video games). To conduct this experiment we progressively move the two moving humans closer to each other and plot the worse of the two motion tracing accuracies as reported by WiDeo in Fig. 10.

WiDeo achieves a motion tracing resolution of 0.5 meters while still achieving an extremely good tracing accuracy of 12cm. So two humans could be standing a little bit more than a foot away from each other (e.g. in a video game), moving their hands closest to each other simultaneously, and still be able to accurately trace their motion.

We also observed that localization error is unaffected; the error is the same as in Fig. 9 . This is expected since the localization technique works by combining measurements from multiple tracing radios when a new moving backscatter component is detected. Since we assume that two human motions do not start exactly at the same time and are usually spaced at least 10ms apart, WiDeo's localization algorithms have a sufficiently long window of time (10ms) in which they can perform localization on a single new object without the presence of a nearby moving object. The same argument applies when the second new motion is detected, by then the first motion is localized and can be accounted for and localization can focus only on the new backscatter components that arise from new moving object.

4.2.5 Impact of number of tracing radios

In this experiment, we see impact on accuracy as we vary the number of radios performing tracing in WiDeo. We conduct the same experiment as in § 4.2.3, but vary the number of tracing radios from one to five. We plot five different CDFs of localization and motion tracing error in Fig. 11 .

As we can see, WiDeo's localization error is poor (4m) with a single tracing radio. This is expected, since WiDeo relies on triangulation to localize well. However motion tracing error is less affected, WiDeo still traces with less than 12cm error. Consequently while we cannot localize with a single radio, we can still trace. The reason is that with a single tracing radio, we cannot get an accurate estimate of the depth (location), but the relative motion from that initial location can still be accurately traced since it only depends on relative shifts in backscatter measurements, which are quite accurate.

Increasing the number of tracing radios helps with localization error, it goes down to 0.7m with five tracing radios. Motion tracing, which is already quite accurate even with a single radio, improves slightly to 7cm. This is expected since triangulation improves with more radios and hence localization improves. However backscatter measurement doesn't depend on having multiple observations, it's done independently by each radio. Hence tracing accuracy only improves by a small amount.

4.2.6 Distinct motions that WiDeo can trace

In this experiment, we check how many independent concurrent human motions can WiDeo trace. We vary the number of human motions occurring concurrently from one to six and plot the median tracing accuracy in Fig. 12.

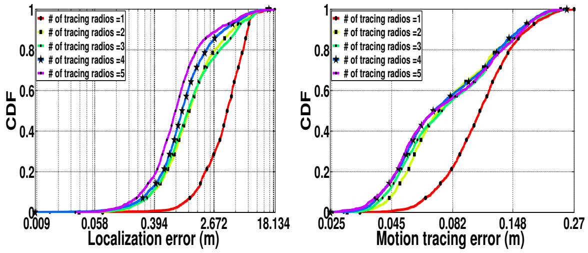


Figure 11: *WiDeo's localization accuracy improves with the number of tracing radios to 0.7m because of better triangulation. However tracing accuracy is unaffected because WiDeo's algorithm's can trace accurately even with information from a single tracing radio.*

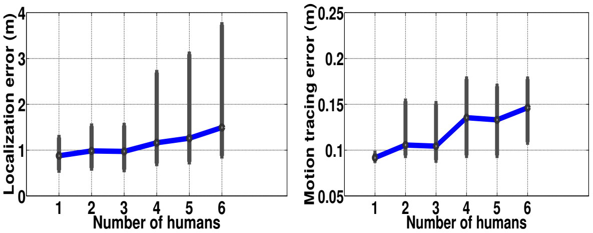


Figure 12: *WiDeo can accurately trace as many as five independent motions that are occurring simultaneously (with error bar representing standard deviation).*

WiDeo can trace up to five concurrent motions with an accuracy of 12cm. To the best of our knowledge, no prior WiFi based system has demonstrated being able to trace five moving humans concurrently. Beyond that accuracy worsens. The reason is that there aren't enough radios to provide sufficient number of backscatter measurements to disentangle these motions. Being able to trace five concurrent motions is sufficient for a home environment, but not for work environments where a far greater amount of motion is expected.

5 Conclusion

This paper demonstrated the surprising capability to build motion tracing camera using WiFi signals as the light source. The fundamental contributions are algorithms that can measure WiFi backscatter and mine them to trace motion. We plan to prototype many interesting applications that builds on top of WiDeo, including gesture recognition, indoor navigation, elderly care and security applications.

References

[1] *David Tse , Fundamentals Wireless Communications.* `http://www.eecs.berkeley.edu/~dtse/Chapters_PDF/Fundamentals_Wireless_Communication_chapter2.pdf`.

[2] *Modeling Indoor Propagation.* `http://www.remcom.com/examples/modeling-indoor-propagation.html`.

[3] *SPEAG Hand.* http://www.speag.com/products/em-phantom/hand/sho-v2-3rb-lb/.

[4] *Wii.* http://en.wikipedia.org/wiki/Wii.

[5] ADIB, F., KABELAC, Z., AND KATABI, D. Multi-Person Motion Tracking via RF Body Reflections.

[6] ADIB, F., KABELAC, Z., KATABI, D., AND MILLER, R. C. 3D Tracking via Body Radio Reflections. In *11th USENIX Symposium on Networked Systems Design and Implementation (NSDI 14)* (Seattle, WA, Apr. 2014), USENIX Association, pp. 317–329.

[7] ADIB, F., AND KATABI, D. See Through Walls with WiFi! In *Proceedings of the ACM SIGCOMM 2013 Conference on SIGCOMM* (New York, NY, USA, 2013), SIGCOMM '13, ACM, pp. 75–86.

[8] ARSLAN, H., ET AL. Channel Estimation for Wireless OFDM Systems. *IEEE Surveys and Tutorials 9*, 2 (2007), 18–48.

[9] BAHL, P., AND PADMANABHAN, V. RADAR: an in-building RF-based user location and tracking system. *Proceedings IEEE INFOCOM 2000. Conference on Computer Communications. Nineteenth Annual Joint Conference of the IEEE Computer and Communications Societies (Cat. No.00CH37064) 2* (2000), 775–784.

[10] BHARADIA, D., JOSHI, K. R., AND KATTI, S. Full Duplex Backscatter. In *Proceedings of the Twelfth ACM Workshop on Hot Topics in Networks* (2013), ACM, p. 4.

[11] BHARADIA, D., AND KATTI, S. Full Duplex MIMO Radios. In *11th USENIX Symposium on Networked Systems Design and Implementation (NSDI 14)* (Seattle, WA, Apr. 2014), USENIX Association, pp. 359–372.

[12] BHARADIA, D., MCMILIN, E., AND KATTI, S. Full Duplex Radios. In *Proceedings of the ACM SIGCOMM 2013 conference on SIGCOMM* (New York, NY, USA, 2013), SIGCOMM '13, ACM, pp. 375–386.

[13] BOYD, S., AND VANDENBERGHE, L. *Convex Optimization*. Cambridge University Press, New York, NY, USA, 2004.

[14] CANDES, E., AND ROMBERG, J. Sparsity and Incoherence in Compressive Sampling, 2006.

[15] CHINTALAPUDI, K., PADMANABHA IYER, A., AND PADMANABHAN, V. N. Indoor Localization Without the Pain. In *Proceedings of the sixteenth annual international conference on Mobile computing and networking* (2010), ACM, pp. 173–184.

[16] CZINK, N., HERDIN, M., AND ZCELIK ERNST BONEK. Number of Multipath Clusters in Indoor MIMO Propagation Environments.

[17] DONOHO, D. L. Compressed sensing. *IEEE Trans. Inform. Theory 52* (2006), 1289–1306.

[18] EKANADHAM, C., TRANCHINA, D., AND SIMONCELLI, E. P. Recovery of sparse translation-invariant signals with continuous basis pursuit. *Signal Processing, IEEE Transactions on 59*, 10 (2011), 4735–4744.

[19] ERCEG, V., SCHUMACHER, L., KYRITSI, P., AND ET AL. TGn channel models. *Tech. Rep. IEEE P802.11, Wireless LANs, Garden Grove, Calif, USA* (2004).

[20] GJENGSET, J., XIONG, J., MCPHILLIPS, G., AND JAMIESON, K. Phaser: Enabling Phased Array Signal Processing on Commodity WiFi Access Points. In *Proceedings of the 20th Annual International Conference on Mobile Computing and Networking* (New York, NY, USA, 2014), MobiCom '14, ACM, pp. 153–164.

[21] GOLDSMITH, A. *Wireless communications*. Cambridge university press, 2005.

[22] GUPTA, S., MORRIS, D., PATEL, S., AND TAN, D. Soundwave: using the Doppler Effect to Sense Gestures. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems* (2012), ACM, pp. 1911–1914.

[23] HARRISON, C., TAN, D., AND MORRIS, D. Skinput: Appropriating the Body as an Input Surface. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems* (2010), ACM, pp. 453–462.

[24] JIA, Y., KONG, L., YANG, X., AND WANG, K. Through-wall-radar localization for stationary human based on life-sign detection. *2013 IEEE Radar Conference (RadarCon13)*, 3 (Apr. 2013), 1–4.

[25] JOSHI, K., HONG, S., AND KATTI, S. PinPoint: Localizing Interfering Radios. In *Proceedings of the 10th USENIX Conference on Networked Systems Design and Implementation* (Berkeley, CA, USA, 2013), nsdi'13, USENIX Association, pp. 241–254.

[26] KIM, D., HILLIGES, O., IZADI, S., BUTLER, A. D., CHEN, J., OIKONOMIDIS, I., AND OLIVIER, P. Digits: Freehand 3D Interactions Anywhere Using a Wrist-Worn Gloveless Sensor. In *Proceedings of the 25th annual ACM symposium on User interface software and technology* (2012), ACM, pp. 167–176.

[27] KOSBA, A., SAEED, A., AND YOUSSEF, M. RASID: A robust WLAN device-free passive motion detection system. In *Pervasive Computing and Communications (PerCom), 2012 IEEE International Conference on* (March 2012), pp. 180–189.

[28] KUHN, H. W. The Hungarian Method for the Assignment Problem. *Naval Research Logistics Quarterly 2* (1955), 83–97.

[29] KUMAR, S., HAMED, E., KATABI, D., AND ERRAN LI, L. LTE radio analytics made easy and accessible. *Proceedings of the 2014 ACM conference on SIGCOMM - SIGCOMM '14* (2014), 211–222.

[30] MAAREF, N., MILLOT, P., PICHOT, C., AND PICON, O. A Study of UWB FM-CW Radar for the Detection of Human Beings in Motion Inside a Building. *Geoscience and Remote Sensing, IEEE Transactions on 47*, 5 (May 2009), 1297–1300.

[31] MALLAT, S., AND ZHANG, Z. Matching Pursuit With Time-Frequency Dictionaries. *IEEE Transactions on Signal Processing 41* (1993), 3397–3415.

[32] MATTINGLEY, J., AND BOYD, S. CVXGEN: a code generator for embedded convex optimization. *Optimization and Engineering 13*, 1 (2012), 1–27.

[33] NARAYANAN, R. M. Through-wall radar imaging using UWB noise waveforms. *Journal of the Franklin Institute 345*, 6 (Sept. 2008), 659–678.

[34] PU, Q., GUPTA, S., GOLLAKOTA, S., AND PATEL, S. Whole-Home Gesture Recognition using Wireless Signals. In *Proceedings of the 19th annual international conference on Mobile computing & networking* (2013), ACM, pp. 27–38.

[35] RALSTON, T., CHARVAT, G., AND PEABODY, J. Real-time through-wall imaging using an ultrawideband multiple-input multiple-output (MIMO) phased array radar system. In *Phased Array Systems and Technology (ARRAY), 2010 IEEE International Symposium on* (Oct 2010), pp. 551–558.

[36] SEIFELDIN, M., SAEED, A., KOSBA, A. E., ELKEYI, A., AND YOUSSEF, M. Nuzzer: A Large-Scale Device-Free Passive Localization System for Wireless Environments. *IEEE Transactions on Mobile Computing 12*, 7 (July 2013), 1321–1334.

[37] SEN, S., CHOUDHURY, R. R., AND NELAKUDITI, S. SpinLoc: Spin Once to Know Your Location. In *Proceedings of the Twelfth Workshop on Mobile Computing Systems &#38; Applications* (New York, NY, USA, 2012), HotMobile '12, ACM, pp. 12:1–12:6.

[38] SEN, S., LEE, J., KIM, K.-H., AND CONGDON, P. Avoiding Multipath to Revive Inbuilding WiFi Localization. In *Proceeding of the 11th Annual International Conference on Mobile Systems, Applications, and Services* (New York, NY, USA, 2013), MobiSys '13, ACM, pp. 249–262.

[39] SHEN, Y., AND MARTINEZ, E. Channel Estimation in OFDM Systems. *Application Note, Freescale Semiconductor* (2006).

[40] TIBSHIRANI, R. Regression Shrinkage and Selection Via the Lasso. *Journal of the Royal Statistical Society, Series B 58* (1994), 267–288.

[41] TIPALDI, G. D., AND RAMOS, F. Motion clustering and estimation with conditional random fields. In *Intelligent Robots and Systems, 2009. IROS 2009. IEEE/RSJ International Conference on* (2009), IEEE, pp. 872–877.

[42] TROPP, J. A., AND GILBERT, A. C. Signal recovery from partial information via Orthogonal Matching Pursuit. *IEEE TRANS. INFORM. THEORY* (2005).

[43] WANG, J., VASISHT, D., AND KATABI, D. RF-IDraw: Virtual Touch Screen in the Air Using RF Signals. In *Proceedings of the 2014 ACM Conference on SIGCOMM* (New York, NY, USA, 2014), SIGCOMM '14, ACM, pp. 235–246.

[44] WELCH, G., AND BISHOP, G. An Introduction to the Kalman Filter. Tech. rep., Chapel Hill, NC, USA, 1995.

[45] WILSON, J., AND PATWARI, N. See-Through Walls: Motion Tracking Using Variance-Based Radio Tomography Networks. *IEEE Transactions on Mobile Computing 10*, 5 (May 2011), 612–621.

[46] XIONG, J., AND JAMIESON, K. ArrayTrack: A Fine-Grained Indoor Location System. In *NSDI* (2013), pp. 71–84.

[47] YOUSSEF, M., AND AGRAWALA, A. The Horus WLAN Location Determination System. In *Proceedings of the 3rd International Conference on Mobile Systems, Applications, and Services* (New York, NY, USA, 2005), MobiSys '05, ACM, pp. 205–218.

[48] YOUSSEF, M., MAH, M., AND AGRAWALA, A. Challenges: Device-free Passive Localization for Wireless Environments. In *Proceedings of the 13th Annual ACM International Conference on Mobile Computing and Networking* (New York, NY, USA, 2007), MobiCom '07, ACM, pp. 222–229.

[49] ZETIK, R., CRABBE, S., KRAJNAK, J., PEYERL, P., SACHS, J., AND THOMA, R. Detection and localization of persons behind obstacles using M-sequence through-the-wall radar, 2006.

[50] ZHANG, D., MA, J., CHEN, Q., AND NI, L. M. An RF-Based System for Tracking Transceiver-Free Objects. In *Proceedings of the Fifth IEEE International Conference on Pervasive Computing and Communications* (Washington, DC, USA, 2007), PERCOM '07, IEEE Computer Society, pp. 135–144.

[51] ZHANG, Z. Microsoft Kinect Sensor and Its Effect. *MultiMedia, IEEE 19*, 2 (Feb 2012), 4–10.

FlexRadio: Fully Flexible Radios and Networks

Bo Chen[†]*, Vivek Yenamandra*[†] *and Kannan Srinivasan*
Department of Computer Science and Engineering
The Ohio State University, Columbus, OH 43210
{*chebo, yenamand, kannan*}*@cse.ohio-state.edu*
†*Co-primary Authors*

Abstract

When a wireless node has multiple RF chains, there are several techniques that are possible; MIMO, full-duplex and interference alignment. This paper aims to unify these techniques into a single wireless node. It proposes to make a wireless node fully flexible such that it can choose any number of its RF chains for transmission and the remaining for simultaneous reception. Thus, MIMO and full duplex are subset configurations in our design. Surprisingly, this flexibility performs better than MIMO or full duplex or interference alignment or multi-user MIMO.

This paper presents the design and implementation of FlexRadio, the first system enabling flexible RF resource allocation. We implement FlexRadio on the NI PXIe 1082 platform using XCVR2450 radio front-ends. FlexRadio node networks achieves a median gain of 47% and 38% over same networks with full duplex and MIMO nodes respectively.

1 Introduction

When a wireless node has multiple radio frequency (RF) chains, the state-of-the-art technology has been to use either all of them for transmission or reception, as in multiple-input multiple-output (MIMO). Recently, many research groups have shown that a node can transmit and receive simultaneously and thus, be full-duplex. Under full-duplex operation, a node activates equal number of RF chains for transmission as it does for simultaneous reception. Thus, when a node has N RF chains, under full duplex, N/2 RF chains are active transmitting RF chains while the remaining N/2 RF chains are receiving RF chains. Under MIMO, all N RF chains are either active transmitting RF chains or active receiving RF chains. There is much work in the wireless community studying which of these techniques are better and *when* [2, 3, 7]. Fundamentally, the capacity achieved by MIMO and full-duplex between a pair of nodes, is the same. The main difference is that MIMO supports N simultaneous transmissions in one direction, while full duplex supports N/2 in both the directions. Still, the total number of transmissions is only N in both the cases[1].

From the above discussion, it is clear that there is no significant difference in the capacity between MIMO and full duplex. However, this paper shows that when we unify MIMO and full duplex, and make the design fully flexible then, surprisingly, the capacity can be improved by 2x when compared to MIMO or full duplex. By flexible, we mean that, out of N active RF chains, our system allows M of them to be transmit RF chains and (N-M) of them to be receive RF chains, where $0 \leq M \leq N$. We call our system, FlexRadio.

Although choosing between MIMO and full duplex configurations gives no improvement in throughput between a pair of nodes, it does improve the overall *network* throughput. This improvement comes from the difference in the interference footprint between MIMO and full duplex, in a network [18, 19]. During a MIMO transmission, a secondary transmission around the receiver and a secondary reception around the transmitter is prohibited. However, a secondary reception around the receiver and a secondary transmission around the transmitter is allowed as long as they do not affect the ongoing transmission. Similarly, during a full duplex transmission, transmission around both the nodes is prohibited, while another reception is possible. FlexRadio's flexibility allows a network to exploit this difference to increase the number of parallel transmissions in a network. Section 3 shows that when every node can choose between full duplex or MIMO operation, the total network throughput increases by 50% compared to the case when all the nodes are either MIMO or full duplex.

[1] Note that in both the cases, a node has 2N (N transmit and N receive) RF chains but, only N of them are active. For rest of the paper, by an N RF-chain node, we imply a node with N active RF chains (either transmit or receive or both) unless explicitly stated otherwise

The gain in FlexRadio is not simply from choosing between MIMO and full-duplex configurations. But, it is from choosing from all available configurations within FlexRadio. Section 3 shows one example where a configuration that is not MIMO or full duplex improves the throughput by 2x, even between a pair of nodes.

Thus, the unified architecture with its adaptability makes it more powerful than the traditional (inflexible) configurations. This is a fundamental improvement in throughput for a multi-RF chain wireless node. Section 3 motivates the need for a flexible architecture and gives some guidelines on choosing the optimal configuration. The optimal configuration depends on the topology, flow demands, wireless channel and the number of RF chains available at other neighbouring nodes.

This paper makes the following contributions;

1. It proposes flexibility as a new radio capability. In Section 3 we motivate this need based on different network properties. Further, we show that FlexRadio can outperform MIMO, full-duplex and interference alignment techniques.
2. It presents the *first* fully flexible FlexRadio prototype. This prototype has multiple novel mechanisms to reduce implementation complexity. First, an antenna placement design that reduces the number of RF cancellation elements needed (Section 4). Second, a novel non-linearity mitigation strategy to reduce complexity of digital cancellation. A naive non-linear elimination technique would require $\mathscr{O}(M^2)$ modules, where M is the number of transmitting RF chains. We eliminate the non-linear components at the transmitter by using a preconditioning module at each transmitter itself. We reduce the number of non-linearity mitigating modules to $\mathscr{O}(M)$ (Section 4).

The flexibility proposed in this paper is a new feature for a wireless node. This has not been studied in information theory or network theory or wireless systems. This new capability has deep implications to wireless networking: A wireless routing protocol can take into account the number of RF chains available at every node and choose the number of RF chains for transmission (and reception) at different nodes so as to maximize end-to-end throughput.

2 A Primer on MIMO and Full Duplex

This section gives a brief overview of capacity, the maximum achievable throughput. The overview helps motivate flexibility as shown in the following section. Capacity is a function of the quality of wireless link. This quality is measured as the ratio between the received signal strength and the local noise at a receiver (SNR).

Since the generic capacity equations are not easy to interpret, often, approximations are used in literature [17]. For the generic case, when node 1 (transmitter) has n_{tx} RF chains and node 2 (receiver) has n_{rx} RF chains, at high SNR, with a well-conditioned channel matrix, the capacity for fading channel is approximated by:

$$C_{High\_SNR} \approx min(n_{tx}, n_{rx}) * log_2(1 + SNR) \qquad (1)$$

Here, the capacity is equivalent to having $min(n_{tx}, n_{rx})$ parallel streams. Thus, at high SNR, the capacity scales linearly with $min(n_{tx}, n_{rx})$ [17].

At low SNR, with a well-conditioned channel matrix, the capacity for the fast fading channel is approximated by:

$$C_{Low\_SNR} \approx n_{rx} * log_2(1 + SNR) \approx n_{rx} * SNR \qquad (2)$$

Here, the capacity is only a function of the number of receive RF chains. It linearly increases with the number of receive RF chains [17]. These approximations are valid for MIMO and full-duplex[2].

Takeaways: When the SNR is high, equalizing the number of transmitting RF chains at the sender and the number of receiving RF chains at the receiver node gives the maximum throughput. When the SNR is low, on the other hand, maximizing the number of receive RF chains at the receiver maximizes the throughput. Note that the low SNR approximation is for very low SNRs (≈ -15dB) at which WiFi node do not operate. However, the intuition applies to SNRs that are reasonably low for WiFi, as shown in Section 5.4.3.

3 The Need for Flexibility

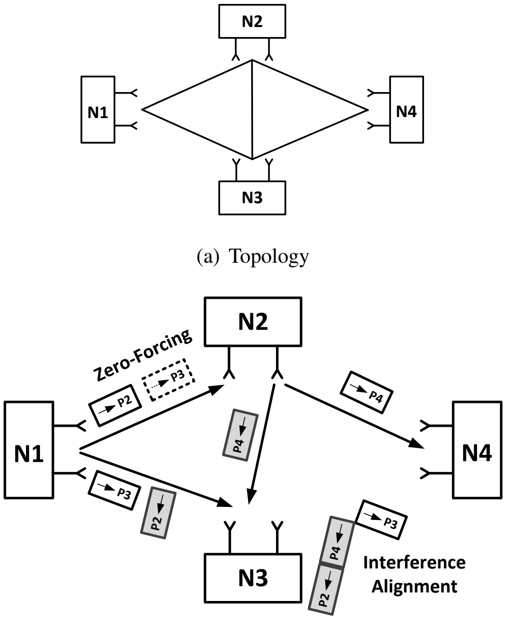
In this section we highlight the benefit of FlexRadio nodes in a wireless network.

3.1 Topology Needs Flexibility

Consider the topology shown in Figure 1(a). It has four nodes with two RF chains each. Nodes N1 and N4 cannot see each other and all other nodes can see each other. This is a common network topology. For example, consider N1 and N4 as APs in an enterprise wireless network that cannot listen to each other. Consider N2 and N3 as clients that can listen to both these APs and to each other. In this topology, if the nodes support fixed MIMO, MU-MIMO or full-duplex functionality, only two packet transmissions can be enabled simultaneously. For example, under MU-MIMO, N1 can simultaneously send one

[2] When multiple RF chains are involved, by full-duplex, we refer to the case that half of the chains are operating as transmitters and the others are receivers.

packet to N2 and another to N3. During this slot, N4 cannot transmit to N2 or N3 to avoid causing interference at these nodes. Similarly, if all the nodes are full-duplex nodes, N1 can send a packet to N2, while N2 sends to N4. At this time N3 cannot transmit as it causes interference at N4. Thus, the maximum number of packets transmitted simultaneously is only two. Thus, enabling a third transmission stream in addition to the two transmission streams causes destructive interference at one of the participating nodes. However, in the above topology, if each node supports flexible functionality, it presents them with the required spatial dimensions (antennas) to explore interference alignment solutions [1, 10] to allow a third simultaneous transmission. It must be noted that interference alignment does not require additional capability for MU-MIMO capable wireless nodes.



(a) Topology

(b) FlexRadio

Figure 1: Topology Needs Flexibility: An example of FlexRadio outperforming MU-MIMO, MIMO and full-duplex, without any flow restrictions. FlexRadio can enable 3 packets to be simultaneously transmitted, while MU-MIMO, MIMO or full-duplex can only enable 2.

In more explicit terms, N1 can send one packet (P2) to N2 and one more (P3) to N3 (as shown in Figure 1(b). Simultaneously, N2 can send a packet (P4) to N4. Since N1 has two antennas, it can null (zero-force) P3 at N2, while aligning P2 with P4 at N3. Since P3 is nulled at N2 and P2 is not, N2 can decode P2. Since N3 is using both the antennas for receiving, it can decode two packets. But, it receives 3 packets. However, since P2 and P4 are aligned, N3 can decode P3 without any interference. At the same time, N4 receives P4 from N2 without any interference. Thus, there are 3 successful packet transmissions. This was possible because of flexibility enabled by FlexRadio and interference alignment techniques. Even when MIMO, MU-MIMO and full-duplex work with interference alignment, they cannot transmit more than 2 packets, while FlexRadio achieves *1.5X* throughput gain.

To understand how FlexRadio was invoked, note that N1 was using its two RF chains to transmit, N2 was using one to transmit and the other to receive, N3 was using both to receive, and N4 was using one to receive. This example shows that FlexRadio can *fundamentally improve capacity of interference limited wireless networks with multi-RF chain nodes.*

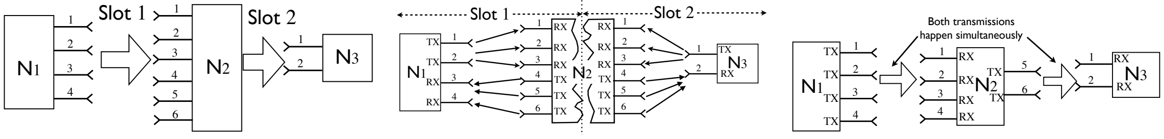
3.2 Flow Demand Needs Flexibility

Performance gains of FlexRadio can be seen in other networks as well. Consider a simple network of 3 nodes; say node 1 has 4 RF chains, node 2 has 6 RF chains and node 3 has 2 RF chains.This is a heterogeneous network with different nodes having different number of RF chains. Assume that each hop has the same, but high SNR. The MIMO scenario is shown in Figure 2(a). In this case, MIMO can support $\frac{1}{2}*4+\frac{1}{2}*2$ parallel streams. Here, the first term corresponds to the performance of the link between node 1 and 2, and the second term corresponds to the link between node 2 and 3. Since only one of the two can be active at any time, their overall performance are scaled by half. From network point of view, three streams are enabled simultaneously.

Note that if full-duplex is used, every node would have to split its RF chains equally to transmit and receive. This is shown in Figure 2(b). For full-duplex also, the number of streams that can be enabled simultaneously is $\frac{1}{2}*4+\frac{1}{2}*2$. The capacity is same as that of MIMO even though the flows are in both directions.

In FlexRadio, however, node 1 can transmit on all 4 of its RF chains and node 2 can receive on 4 RF chains. Simultaneously, node 2 can forward packets using the remaining 2 RF chains to node 3, while node 3 uses all of its RF chains for receiving. This is shown in Figure 2(c). Here, node 2 is able to transmit (forward) while receiving because FlexRadio supports full-duplex operation. Now, the number of stream supported in this setup is $4+2$. As before, the first term is for the link between node 1 and 2, and the second is between node 2 and 3. There is no scaling for these quantities because these flows happen simultaneously. Therefore, the combined system can support 6 streams. This is *twice* as much as a traditional MIMO or full-duplex system.

However, when the PHY is MU-MIMO capable (such as APs for 802.11n), the same capacity as FlexRadio can



(a) MIMO (b) Full-Duplex (c) FlexRadio

Figure 2: A Heterogeneous Network with different nodes having different number of RF chains.

be achieved where Node 2 uses 4 RF chains to transmit to Node 1 and the remaining to transmit to Node 3 simultaneously. However, when there is a desired flow demand, say Node 1 to Node 2 to Node 3, FlexRadio can improve the throughput of a MU-MIMO system. For this flow demand, the MU-MIMO operation does not provide over MIMO operation.

3.3 Channel Needs Flexibility

Consider nodes 1 and 2 each with M RF chains. Assume, both of them want to transmit to each other. Also, assume a very low SNR channel.

When MIMO alone is used, Node 1 uses all M RF chains to transmit, while Node 2 uses all M RF chains to receive. In the low SNR region (for poor channel conditions), the capacity is simply proportional to the number of receivers used, as shown in Equation 2. Thus, the capacity is $C_{MIMO} \approx M * SNR$.

When fullduplex is used, node 1 uses $\frac{M}{2}$ RF chains to transmit and $\frac{M}{2}$ RF chains to receive, same as Node 2. In this case, we compute the capacity for both transmission directions. The total capacity in the low SNR regime is $C_{FD} \approx \frac{M}{2} * SNR + \frac{M}{2} * SNR$. This capacity is same for both MIMO and full-duplex.

When the flexibility is provided, nodes 1 and 2 can choose the number of RF chains they wish to transmit and receive over. Note that, at low SNR, the nodes should maximize the number of receive RF chains. Therefore, when nodes 1 and 2 use only one RF chain to transmit and the remaining (M-1) RF chains to receive, the sum capacity, in the low SNR region, is $C_{FlexRadio} \approx (M-1) * SNR + (M-1) * SNR$. This is almost double the sum capacity compared to MIMO and full-duplex.

In all these examples, we assumed a central node is made aware of the RF resources of all nodes in the network and their respective traffic demands. We discuss the MAC implications briefly in Sec. 7. In summary, flexibility enables FlexRadio nodes to achieve significant performance gains based on topology, flow and channel constraints.

4 Design Overview

Based on FlexRadio's configuration, the self-interference constituents change. A FlexRadio *self-interference* cancellation circuitry should hence support all these configurations. The challenge in designing FlexRadio's self-inteference cancellation circuitry is the following. It should include cancellation circuitry that accounts for every TX RF chain, a potential source of self-interference, at every RX RF chain. This leads to $M * (M-1)$ cancellation circutry elements for an M RF chain system. This can make implementing FlexRadio node highly expensive. This section presents a design that significantly reduces the number of cancellation elements. For example, for a four RF-chain FlexRadio, our design only requires 2 elements, while the naive approach needs 12.

A self-interference channel between two antennas consists of two components at RF frequencies: line-of-sight and non-line-of-sight component. The line-of-sight component of the interference is simply a function of the distance between the two antennas.[3]This component can be estimated and accounted for using free-space path loss equations. The non-line-of-sight component is a function of the environment. The transmitted signal can reflect off objects in the environment and contribute to the self-interference at the receiver. We account for the self-interference in two stages. In the first stage, majority of the line-of-sight self-interference component is accounted for by RF cancellation (Sec. 4.1). The residual self-interference including the entire non-line-of-sight component is accounted for by digital cancellation (Sec. 4.2).

Finally, a recent work showed that self-interference has non-linear components due to the power amplifier [4] that needs to be accounted for. Extending their non-linear mitigation strategy to an M RF chain system naively requires $\mathcal{O}(M^2)$ non-linear mitigation modules. This section presents a technique that reduces this number to $\mathcal{O}(M)$.

[3] Assuming omni-directional antennas and no obstruction between the two antennas.

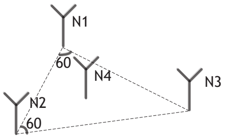


Figure 3: Antenna placement for a four RF-chain FlexRadio system; Three antennas are placed on the vertices of an equilateral triangle with N_4's antenna placed on the centroid

4.1 RF Cancellation

RF cancellation circuitry accounts for the line-of-sight component of self-interference. This component of self-interference signal typically experiences delay and attenuation that is *only* a function of the distance between the TX and RX antenna. Every such link between a transmit and receive RF chain in a FlexRadio node needs a self-interference cancellation block that matches the delay and attenuation experienced by the self-interference over air. We refer to this block as the *delay and attenuation block.* To design an efficient self-cancellation circuitry, we propose an antenna placement scheme that leverages its geometrical symmetry to alleviate the complexity of the RF cancellation circuitry. Symmetric antenna placement makes it possible to combine multiple self-interference signals that have the same delay and attenuation. By doing so, the combined self-interference needs only one *delay and attenuation* block. It must be noted that while the line-of-sight component has the same delay and attenuation as long as the distance between the transmit and receive antenna is the same. the multipath (non-line-of-sight) component can be different. However, our experiments (in Sec. 5) show that these multipath components are not as large as the line-of-sight component and therefore, can be cancelled in the digital domain (explained in the next subsection).

4.1.1 Antenna Placement Scheme (APS)

Figure 3 illustrates the antenna placement scheme for a four RF-chain FlexRadio node. Three antennas, N_1, N_2 and N_3, are on the vertices of an equilateral triangle with the fourth antenna, N_4, at the centroid. In addition to placing the antennas as illustrated, we define an order in assigning which RF-chain to transmit (receive) for a given configuration of FlexRadio. The order of transmission for a four RF-chain FlexRadio node in descending order is: N_1, N_2, N_3 and N_4. For example, N_1 is assigned as the only transmitter when FlexRadio is configured in (1/3) mode[4]. The *advantage* of biasing the order of transmission (reception), together with the symmetry of the proposed antenna placement scheme is the following: *The attenuation and delay of the transmitted signal at a given receiver is independent of the transmitter chain.* In other words, the *delay and attenuation block* in the cancellation path of a given receiver is decoupled from the configuration of the FlexRadio node. For example, the attenuation and delay of the *self interference* signal at N_4 is the same whether originating from N_1, N_2 or N_3. This is true because of biasing the transmission order as this eliminates the possibility of a *self-interference* signal at N_2 or N_3 to originate from N_4.

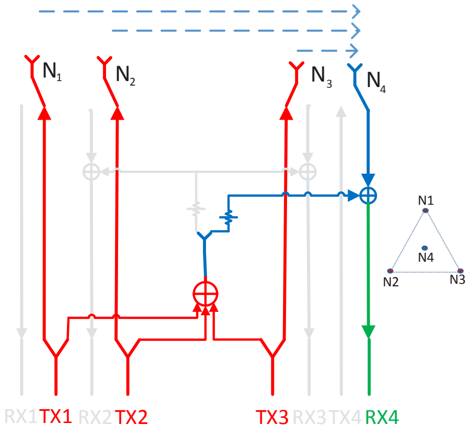


Figure 4: Simplified block diagram of the RF cancellation circuit for a four RF-chain FlexRadio. N_1,N_2,N_3 and N_4 are the 4 antennas with associated TX/RX chains. The figure highlights the active paths in the *self interference* cancellation circuitry for a 3/1 configuration. The cancellation signals from TX1, TX2 and TX3 are combined, inverted (π phase shifter not shown in figure for simplicity) and fed through the *delay and attenuation block* associated to receiver RX4. The *delay and attenuation block* matches the identical attenuation and delay of the *self interference* signals. The dashed lines directed from the TX antennas to the RX antennas illustrate the link in air traversed by the *self interference* signals. The top view of the antenna placement scheme is shown next to the block diagram.

4.1.2 Cancellation Design

Figure 4 illustrates a simplified block diagram of the *self-interference* RF cancellation circuitry for a four RF-chain FlexRadio node. It illustrates the RF signal paths connecting the antennas with the respective RF chain. Specifically, it highlights the *active RF paths* when the

[4] We define a configuration, n_t/n_r, of FlexRadio as a mode of operation in which it commits n_t of its RF-chains to transmit and the remaining n_r RF-chains to receive simultaneously.

node is configured in 3/1 mode. The *inactive* RF paths are greyed. The notation for the antennas in Figure 4 is consistent with that in Figure 3. The TX/RX RF chains are labelled as TXi/RXi respectively, where i is the index of the associated antenna.

As illustrated in Figure 4, in the 3/1 mode, the switches on antennas N_1, N_2 and N_3 are toggled towards the transmit RF chains TX1, TX2 and TX3 respectively, while switch on antenna N_4 is toggled towards the receive RF chain, RX4. This is in accordance with the transmission order given in Sec. 4.1.1. We explain the cancellation circuitry design by first looking at the active RF paths from the transmit RF chains and then the active RF paths to the receive RF chains. Specifically we will consider 3/1 scenario illustrated in Figure 4, to underline *how* our symmetric antenna placement design enables us to reduce complexity of the design.

The TX Chains. As indicated in Figure 4, the power from each of the Tx chains, $TX1$, $TX2$ and $TX3$, is split into two paths - *transmit path* and *cancellation path.*

The *transmit path* from each TX chain feeds the power to its corresponding antenna. As indicated in Figure 4, the path from $TX4$ to the switch is *not* split. In other words, there is not cancellation path from TX4. This is because of the biasing order in Sec 4.1.1. When N_4 is the transmitter, FlexRadio is configured as 4/0 and thus the FlexRadio node has no active receive RF chains and thus no self-interference.

The *cancellation paths* from the TX chains feeds part of the power to the receive RF paths to enable self-interference cancellation. Self-interference cancellation at a given receiver is achieved by subtracting the self-interference signal it receives (on its antenna) with an exact copy of it. The cancellation path is responsible for *generating an exact copy of the self-interference signal* to each receive RF path. We call this the cancellation signal. The cancellation path draws part of the transmit power to generate a copy of the transmitted signal. This cancellation signal is then subjected to delay and attenuation to match that experienced by the self-interference over the air.

Exploting Symmetric Antenna Placement and Biased Transmission Order: Symmetric antenna placement coupled with transmission biasing order decouples the self-interference channel at a given receiver from the potential source of self-interference. For example, the delay and attenuation of the self-interference channel at receiver N_4 is the same irrespective of whether the source of self-interference is N_1, N_2 or N_3. This allows us to combine the cancellation signals and subject the combination of these cancellation signals to a *delay and attenuation block* that matches that experienced at that receiver[5]. Thus, as indicated in Figure 4, the cancellation signals from TX1, TX2 and TX3 are combined and are collectively subjected to match the delay and attenuation experienced at receiver N_4.

The receiver's perspective. As indicated in Figure 4 each RF path between the RF switch and the receivers RX2, RX3 and RX4 has a combiner. The combiner adds the received signal from the antenna with the inverted copy of the generated cancellation signal to implement self-interference cancellation in the RF domain. Consider $RX4$. $RX4$ is subject to *self interference* from N_1, N_2 and N_3. One input to the combiner in the RF path from N_4 to RX4 is the signal received by the antenna, N_4, itself. This signal is a combination of self-interference and the desired signal intended for the receiver RX4. The other input is the internally generated inverted copy of the combined self-interference signal as discussed previously. Thus, ideally at the combiner output, while the desired signal passes through unchanged, the self-interference signal received at the antenna is cancelled by its internally generated inverted copy.[6] As an aside, $RX1$ does not need a combiner in its path since when N_1 is the receiver, so are all the other RF-chains of the FlexRadio node.

***Delay and Attenuation Block*: Beneath the abstraction.** Each *delay and attenuation block* consists of a variable attenuator and a variable delay block that are controlled by from the baseband. By controlling the attenuator and the phase shifter, the cancellation signal can be conditioned to be an inverted replica of the signal received at the corresponding receiver.

Finally, the switch, illustrated in Figure 4 is used to connect either TX or RX path to the antenna depending on the configuration of the RF-chain. Figure 4 illustrates the active signal paths when FlexRadio is configured as 3/1. For example, when changing from mode 3/1 to mode 2/2, the RF switch associated with N_3 switches to the receive RF path. Simultaneously, TX3 is deactivated while RX3 is activated. Deactivating TX3 renders its corresponding transmit and cancellation paths in the cancellation circuitry inactive. At the same time, the RF path from N_3 to RX3 is active with its associated combiner and *delay and attenuation block.*

Is the symmetry assumption realizable? The requirement of high self-interference cancellation required ($\approx 110dB$) implies that the symmetrical placement is strictly observed. For this, we need to ensure that the omni-directional antennas are parallel to each other and are exactly placed as indicated in Fig. 3. We implement the cancellation circuitry on a PCB and couple the antennas

[5] Before passing the combined signal through the *delay and attenuation block*, we invert the signal to enable subtraction at the receiver using just a combiner

[6] This is called RF cancellation since the self-interference cancellation is performed completely in the RF domain.

to the PCB using SMA cables. Existing PCB manufacturing tolerances enable us to place objects on the PCB within an accuracy of 2 mils (1 mil = $\frac{1}{1000}$ inch). While the antennas are not perfectly omni-directional, we observe that inaccuracy in this modeling is accounted in digital cancellation where the self-interference channel is explicitly measured.

4.2 Digital Cancellation

Digital cancellation is used to capture the multipath components of the self-interference. The self-interference from equidistant transmit antennas to a receive antenna likely experience different multipath profiles. Our digital cancellation design is similar, in principle, to previously proposed techniques [6, 16]. This cancellation module estimates the coefficients of the multipath components using a *finite impulse response* (FIR) filter. Unlike the RF cancellation technique, an M RF chain FlexRadio system needs $M*(M-1)$ FIR-based digital cancellation modules. However, joint channel estimation techniques have been proposed to reduce the complexity of the digital cancellation implementation [3]. These techniques can be applied here as well to reduce resource utilization of digital cancellation implementation.

A recent work [4] showed that FIR-based digital cancellation alone does not suffice to achieve the 110dB total cancellation needed for a WiFi full duplex system. This work identified non-linear components of self-interference that cannot be estimated using FIR filters. It proposed modeling the non-linear component using a polynomial function at each receiving RF chain to mitigate its effect. Thus, in an M RF-chain FlexRadio node, each receiver models the non-linearities of M-1 transmitters. Since every antenna can be configured as a receiver, we would require $\mathcal{O}(M^2)$ such modules.

Can we reduce the number of non-linear mitigation modules from $\mathcal{O}(M^2)$? We present a technique to reduce this number from $\mathcal{O}(M^2)$ to $\mathcal{O}(M)$. The key insight here is that the non-linear components arise from the transmit RF chain's power amplifier [4]. Therefore, instead of estimating and correcting for this non-linearity at the receiver RF chain, we estimate it at the transmitter RF chain and correct for it even before transmission. This *pre-conditioning* needs to be done only at the transmit RF chains. This reduces the complexity from $\mathcal{O}(M^2)$ to $\mathcal{O}(M)$. While joint channel estimation techniques have been proposed to further reduce the complexity of digital cancellation implementation [3], decoupling the digital cancellation and non-linear mitigation from FlexRadio's configuration assists in supporting the flexibility desired.

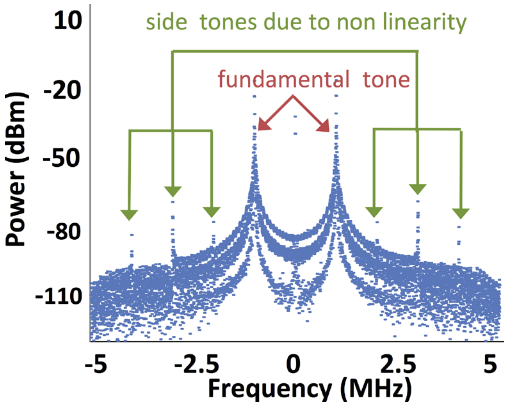


Figure 5: The effect of non-linearity on the transmitted PSD. In addition to the fundamental tones, the side tones prop up due to non-linearity of the transmitter.

4.2.1 Dealing with non-linearities

The distortion caused by transmitter non-linearity on the transmitted signal is illustrated in Fig. 5 when the transmitter sends two single tone frequencies. Similarly, for a wideband OFDM type symbol, the non-linearity results in increased power in the side-bands (adjacent band).

The observed non-linearity can be understood by looking at the received signal (without pre-conditioning):

$$Y(x) = \sum_i \alpha_i x^i \qquad (3)$$

where x is the voltage of the analog signal input to the power amplifier. This simple model models the power amplifier non-linearity using a polynomial. Estimating the non-linearity is equivalent to finding the coefficients of the polynomial. Contrary to the technique proposed in [4], we tackle this phenomenon by *pre-conditioning* the input signal of the power amplifier at transmitter itself.

Thus, instead of transmitting the signal x, we transmit the following,

$$f(x) = \alpha_1 (x - \sum_{i=3,5,7,9,11} (\alpha_i/\alpha_1) x^i) \qquad (4)$$

Thus, when the input signal is preconditioned, the output of the power amplifier is approximately linear. In effect, the signal preconditioning block lowers the input signal power to the power amplifier thus preventing its high gain from saturating the output, thus reducing nonlinearities.

Will the non-linearity introduced in Eq. 4, violate linearity assumptions of communication systems design? It must be noted that here, we introduce preconditioning at the signal level in an effort to *balance* the nonlinearity of the power amplifier and make the resulting output signal linear. This is equivalent to preconditioning

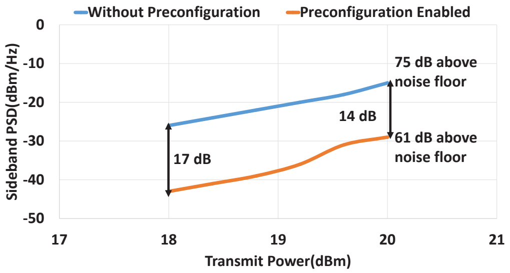


Figure 6: The PSD of the transmitter sidebands reduces after enabling the preconfiguration module.

the signal at the receiver side *after* the signal experiences non-linearity of the power amplifier. The preconditioning in effect, reduces the power of the non-linear components in the channel and makes the linear approximation of communication systems more valid.

We model the non-linearity of the transmitter in the training phase. We send a training series of analog inputs of known power to the power amplifier and derive the coefficients of the polynomial by measuring the output power. Once we model the non-linearity, we precondition the signal using equation 4. We transmit a wideband OFDM signal by sweeping the transmit power from close to its maximum power to its maximum power. When transmitting this OFDM signal, we measure the power of the sidebands, when the preconfiguration module is disabled and again when the preconfiguration module is enabled. We use an external power amplifier to boost the power up to 20 dBm.

Figure 6 plots the findings from our experiment. We vary the transmit power from 18dBm to 20 dBm. This power range captures the strongest non-linear behaviour of the power amplifier. The preconditioning module decreases the PSD of the sidebands by 17dB at transmit power of 18dBm and by 14 dB at the highest transmit power. The decrease in reduction of the PSD of the sidebands at higher power suggests that the fundamental tone is more saturated, i.e. the power amplifier exhibits a stronger non-linear characteristic. However, across the entire power range of the transmitter, enabling the preconfiguration module limits the PSD of the sidebands to at most **61 dB** above the noise floor at the receiver.

5 Implementation and Evaluation

The antenna placement design assumed that its symmetric design implied equal attenuation and delay for line-of-sight self-interference from equidistant transmit RF chains. This lead to the reduction in the number of programmable attenuators needed for RF cancellation. When this assumption does not hold, cancellation performance degrades potentially below the 110dB cancellation needed for WiFi. This section evaluates the design principles presented in the previous sections. We achieve the desired 110dB cancellation with our design.

5.1 FlexRadio Implementation

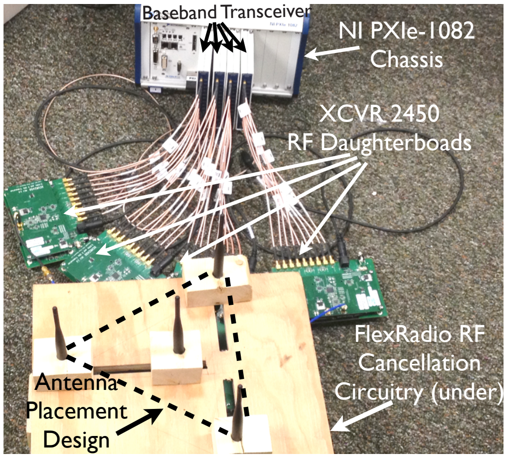

Baseband Transceiver
NI PXIe-1082 Chassis
XCVR 2450 RF Daughterboads
FlexRadio RF Cancellation Circuitry (under)
Antenna Placement Design

Figure 7: Four RF-chain FlexRadio system

Figure 7 shows our four RF-chain FlexRadio system implementation. It can be viewed as a cascade of three high-level modules connected to each other using SMA cables: *The Antenna Placement site, RF cancellation circuitry, RF/baseband chains*.

The antennas are held in position by sliding them through slotted wooden blocks. They are connected to the cancellation circuitry using SMA cables. The distance between the antennas on the vertices and the centroid antenna is set to 5.5”.

The RF chains are implemented using the XCVR 2450 (RF front end), the NI-5781 (data converter module with a 14 bit ADC and 16bit DAC) and the NI PXIe-7965R (a Xilinx Virtex-5 based FPGA) for baseband processing including digital cancellation implementation. The FPGAs are housed in a chassis that contains communication and clock backplanes to facilitate synchronization and communication among the FPGAs.

Figure 8 shows the designed FlexRadio RF cancellation circuitry. The TX and RX ports labelled in Figure 8 are consistent with the labelling used in Figure 4. The cancellation circuit employs the PE43704, a 0-31.75dB attenuator that can be programmed in 0.25dB steps. The attenuators are controlled with on-board switches. We match the delay between the cancellation and self-interference paths with a symmetrical copper trace design on the PCB board. We built the circuit on Rogers 4350 PCB material. The board dimensions are 9”x8”.

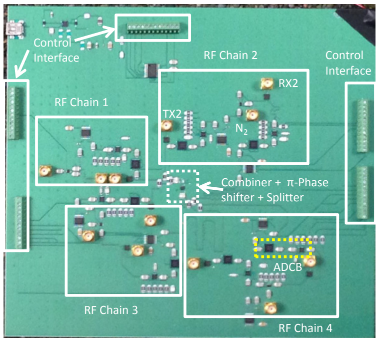


Figure 8: FlexRadio RF cancellation circuitry. Each RF chain contains three ports: Antenna, TX and RX port (indicated in Figure 4)The block labelled ADCB is the *delay and attenuation* block described in Section 4

5.2 Self-Interference Cancellation Evaluation

FlexRadio's self-interference cancellation has three distinct modules: RF, digital cancellation module and the transmitter preconditioning module. These modules, in unison, enable FlexRadio to nullify its self-interference in each of its operating configurations.

The RF cancellation cancels the line-of-sight component of the self-interference. Digital cancellation module estimates the channel and nulls the multipath component of self-interference. However, the digital cancellation module cannot predict the non-linearity of the transmitter. As indicated in Sec. 4.2.1, the preconditioning module limits the power in the sidebands to 61 dB over the noise floor. Thus, FlexRadio needs to provide RF cancellation of at least 61 dB to eliminate the non-linear components introduced by the transmitter.

Is the symmetric design effective? We evaluate the self-interference cancellation of FlexRadio over all of its operating modes. We place our four RF-chain FlexRadio prototype inside our lab - a typical indoor environment with metallic cubicles and furniture. We transmit 20 MHz OFDM signal at the transmitters in each of these modes. Figure 10 illustrates the PSD at the centroid at different stages of self-interference cancellation for different configurations of FlexRadio. The RF cancellation at the centroid is constant across different modes of operation and is 68 dB. As illustrated in Figure 10, this is sufficient to reduce the power in the side bands (and thus significant portion of the non-linear component) to the noise floor. The RF cancellation is a function of only the antenna placement (since we do not place any objects between the antennas) and we observe it to be at least 68 dB at all RF chains in our prototype.

Evaluating Digital Cancellation Effectiveness: Digital cancellation effectiveness relies on the accuracy of the self-interference channel estimation. Intuitively, measuring the channel response over a longer duration helps in estimating the channel better. Fig. 9 illustrates the digital cancellation performance as a function of the channel estimation time. As seen in Fig. 9, for a channel estimation time of 7.4μ seconds, 42 dB of digital cancellation is achieved. Our digital cancellation module cancels the residue signal from RF cancellation down to the noise floor for all operating modes of FlexRadio.

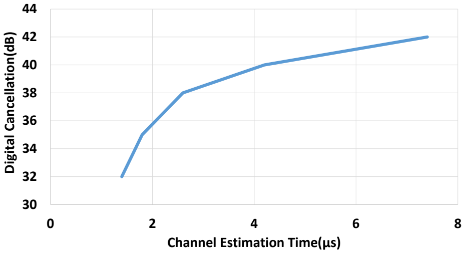


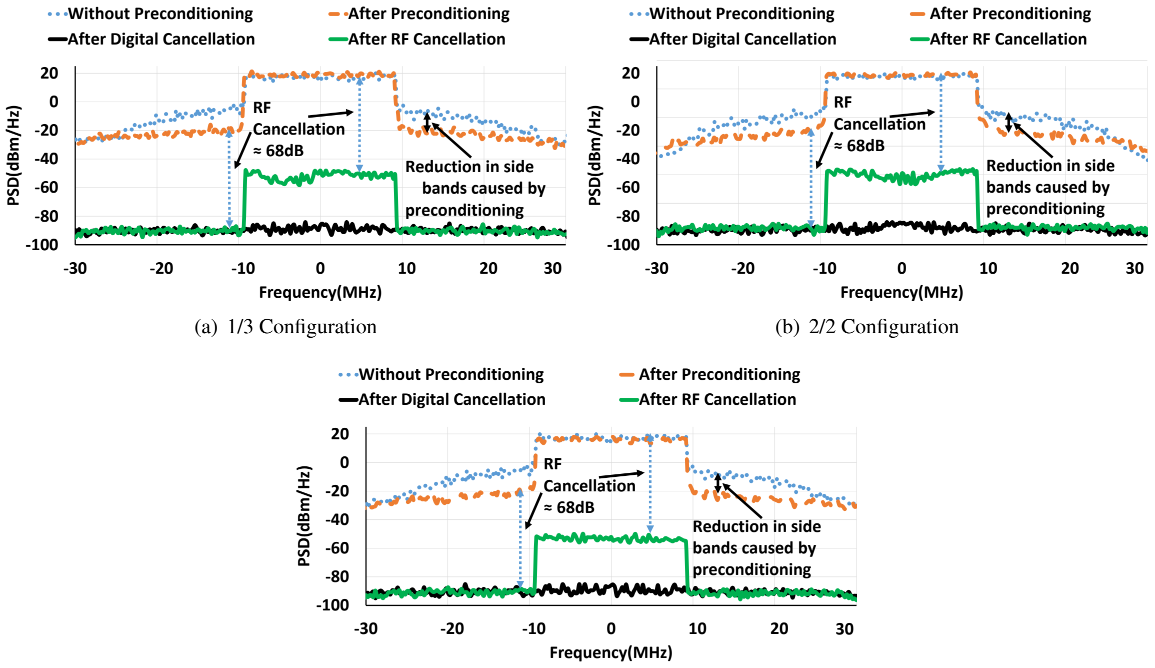
Figure 9: Digital Cancellation as a function of time taken to estimate the self-interference channel.

Figure 10 explicitly illustrates the spectrum at the centroid antenna after RF cancellation. When FlexRadio is operating in mode, 1/3, the effect of multipath is more pronounced after RF cancellation indicated by the trough in the residual spectrum after RF cancellation. However, the depth of this trough decreases as the number of transmitters increases i.e the effect of multipath is lesser. In the mode 3/1, the spectrum after the RF cancellation is almost flat. This is because when the number of transmitters increases, the multipath component decreases as the number of line-of-sight components increase.

The RF cancellation at the centroid includes 26 dB attenuation of the self-interference signal over air. Due to space constraints, the power spectral density at each of the other vertices is not included. The RF cancellation at the vertices is 70 dB, due to the the increase in attenuation of the self-interference over the air (FlexRadio's priority ensures that a receiver at the vertex experiencing self-interference only from transmitters positioned at other vertices of the equilateral triangle).

5.3 Configuration Switching Time

When switching from one FlexRadio configuration to another, the switching time can include the time needed for carrying out some, if not all, of the following events: Switching of the RF switches to change receive chains to transmit chains or vice versa; Channel estimation between all the transmit and receive links in the baseband - this event loads the coefficients of the FIR filters used to model the self-interference channels required for digital cancellation; Switching the baseband state to make



(a) 1/3 Configuration

(b) 2/2 Configuration

(c) 3/1 Configuration

Figure 10: PSD at the centroid for different operating modes of FlexRadio. Preconditioning reduces non-linear components, RF cancellation achieves 68dB cancellation, FIR-based digital cancellation brings the remaining self-interference to the noise floor achieving a fully working FlexRadio.

the additional transmit (receive) FIFO available (For instance, when switching from mode 2/2 to 3/1 an additional transmit FIFO is required). Explicitly, to switch between transmission modes 4/0 and 0/4, FlexRadio only needs to switch the RF switches at each RF chain from the transmit RF chain to the receive RF chain. However, when FlexRadio switches from mode 0/4 to mode 3/1, all the events listed above have to be accomplished to transition between the two modes.

The switching and settling times of the programmable attenuator used in FlexRadio are 1.1μs and 2μs respectively. The symmetric antenna placement of FlexRadio decouples the *delay and attenuation block* at each receiver chain from the configuration of FlexRadio. Thus, switching between different configurations of FlexRadio does not require reprogramming the attenuator. None the less, the preconfiguration module and the attenuators used in RF cancellation are tuned periodically to account for changes in circuit behavior due to change in temperatures, humidity etc. However, these tuning requirements are independent from switching FlexRadio configurations and are infrequent.

In our implementation, the maximum switching time occurs when FlexRadio switches from transmission mode 0/4 to 3/1, as the digital cancellation module needs to estimate three channels - between three transmitters to the receiver - in a sequential manner. As indicated in Figure 9, channel estimation time of 7.4μs yields 42dB of digital cancellation in our implementation. Thus the total time to estimate all the channels when FlexRadio switches to 3/1 transmission mode is $\approx$ 22.5μs. The switching time for off-the-shelf RF switches is of the order of tens of nanoseconds. Further, the time to make the required FIFOs available (either a transmit FIFO or a receive data) is of the order of hundreds of nanoseconds. Thus, the maximum time to switch between different transmission modes of FlexRadio is within **25 μs**.

Is the switching time overhead significant? FlexRadioconfiguration changes are motivated by changing topology or flow constraints. Many factors can affect flow constraints. Typical channel coherence time is an ultra-agressive rate estimate of changing topology constraints. However. coherence times even for mobile channels can be hundreds of milliseconds. Thus, under most circumstances, switching between different FlexRadio configurations presents negligible overhead.

5.4 FlexRadio in a network: Experiment setup and evaluation

Having evaluated the effectiveness of FlexRadio's self-interference cancellation strategies and its configurability, in this section, we evaluate the performance of FlexRadio nodes in a network. We perform a set of experiments using different network topologies, flow constraints and channel conditions. We compare the performance of FlexRadio nodes in these networks with the performance of wireless nodes having a fixed functionality (MIMO, full duplex and Multi-User MIMO (MU-MIMO)) in these networks. For fixed full-duplex radios mentioned in this section, half of their RF chains are used for transmission while the rest are assigned for signal reception. So we refer to these as half-half full-duplex.

All modes of radio operation, i.e. FlexRadio operation or fixed function, use standard modulation and coding schemes of WiFi's 802.11g transmissions; 1/2 BPSK, QPSK, QAM16 and QAM64, 2/3 BPSK, QPSK, QAM16 and 3/4 QAM64. All the experiments are conducted in the 2.4GHz ISM band over a bandwidth of 20MHz. Theoretically, FlexRadio nodes should be able to operate on different frequencies as it is based on the symmetry components placement. However, due to the manufacturing limitation of the frequency selective RF components on our PCB board (programmable attenuator, balun and switches), we operate in the 2.4GHz band for which these components have been designed.

5.4.1 FlexRadio in Interference-limited Networks

We evaluate the performance benefits of FlexRadio nodes in interference limited networks as discussed in Section. 3. For this experiment, we place four wireless radio nodes according to the topology as shown in the Figure 1(a). Each radio is implemented on the NI software radio defined platform described in the previous section. For this topology, we compare the performance of FlexRadio nodes with MIMO and half-half full duplex nodes. In both MIMO and full-duplex networks, enabling any two transmission streams simultaneously causes interference at the remaining passive nodes thus preventing another transmission stream. FlexRadio nodes can be configured to make the necessary spatial dimensions (antennas) available to align interference to enable a third stream.

When evaluating FlexRadio nodes in this topology, all nodes compute their channels to neighboring nodes (For instance, node N1, in Figure 1(a), computes the channel between itself and nodes N2 and N3 and so on.). This is required to implement interference alignment. In our implementation, the nodes share the computed channel information over Ethernet. Further, we use the communication backplane of our NI platform to synchronize the distributed nodes in time. There are other techniques in literature to achieve the same requirement [15, 20]. We transmit 200 packets over each enabled transmission stream. We measure the throughput over all active links at their highest possible data rates. We repeat this experiment for 50 different locations of nodes N2 and N3.

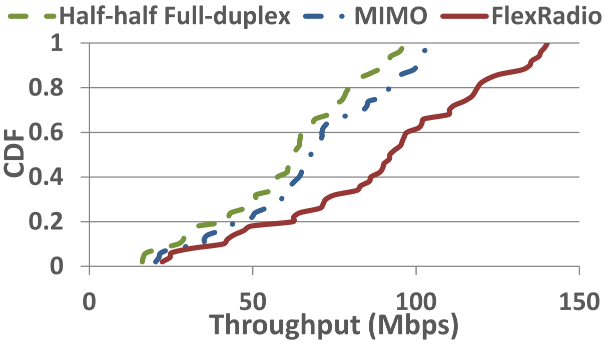


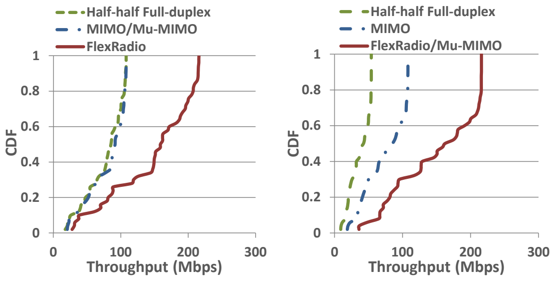
Figure 11: Throughput comparison between MIMO, fixed full-duplex and FlexRadio for the topology shown in Figure 1(a)

Figure 11 plots the CDF of the throughput measured at these locations. FlexRadio outperforms full-duplex and MIMO performance by **47%** and **38%** respectively. This is slightly below the 50% gain anticipated in Section 3. The slight drop in gain can be attributed to the additional channel measurement required between nodes N2 and N3. None the less, the gain is significant over existing MIMO and full-duplex technologies without requiring significant hardware overhead (over full-duplex nodes) or configuration switching overhead.

5.4.2 Adjusting Configuration Based on Flow Demand

We evaluate the benefits of flexibility in networks with flow constraints. We perform this experiment in a three-node network. The radio in the middle has four RF chains. The other two radios with two RF chains cannot hear each other (similar to the topology in Figure 2(a)). We repeat the experiment at 50 different locations to capture different channel conditions. The flow constraint is defined similar to that in Figure 2(a). We measure the throughput for each experiment in a method similar to that described in the previous subsection. We compare the throughput of the network between fixed full-duplex, MIMO, MU-MIMO and FlexRadio nodes. Figure 12(a) plots the CDF of the throughput. We can see that, as expected in section 3, when the middle node operates under 2Tx/2Rx FlexRadio configuration and the other two nodes operate as MIMO receiver (0/2) and MIMO transmitter (2/0) separately the optimal network throughput is achieved. This configuration achieves *twice* the throughput of the other configurations. Note that, for this flow constraint, MU-MIMO does not outperform MIMO.

We repeat the experiment for each of these 50 locations. However, this time we have the middle four RF



(a) (b)

Figure 12: In the flow demand (a), the four RF chain radio wants to receive some packets from one 2-RF chain radio and transmit to another one. In (b), all the two 2-RF chain radios want to transmit to the middle one.

chain node receive from the other two nodes all the time. We plot the CDF of the throughput distribution for this flow constraint in Figure 12(b). In this scenario, MU-MIMO presents the throughput optimal solution, which is the configuration that FlexRadio adopts. Through this experiment, we verify that FlexRadio enables each node in a network to *adapt* to a configuration that achieves optimal network performance.

5.4.3 Varying Channel Conditions

Finally, we seek to evaluate FlexRadio nodes in different channel conditions. Theoretically, it has been deduced that when the SNR of the channel is low, maximizing the number of RF chains/antennas at the receiver maximizes the throughput [17]. However, in the theoretic perspective, this phenomena is observed at really low SNR (around -20dB), where WiFi transmission does not occur.

None the less, we perform an experiment where two radios with four RF chains wish to transmit to each other. Under a reasonable WiFi channel, the SNR varies around 5dB. At this SNR, all the radios choose the lowest data rate (5.5Mbps) corresponding to 1/2 BPSK. For this experiment, we run MIMO in two configurations: One Stream MIMO and Two Stream MIMO. Under One Stream MIMO, all of the transmitting RF chains send the same data. This is usually the optimal strategy under very low SNR conditions. The Two Stream MIMO is the typical MIMO configuration where two RF chains send different data streams, the normal MIMO operation.

We measure the throughput for this scenario for half-half full-duplex, the two MIMO configurations and FlexRadio (1/3) configuration.

The experiment is repeated 50 times and the CDF of the throughput is plotted in Figure 13. Surprisingly, FlexRadio outperforms other configurations ≈ 85% of the times even when the channel SNR varies around 5dB. On some instances one stream MIMO performs better.

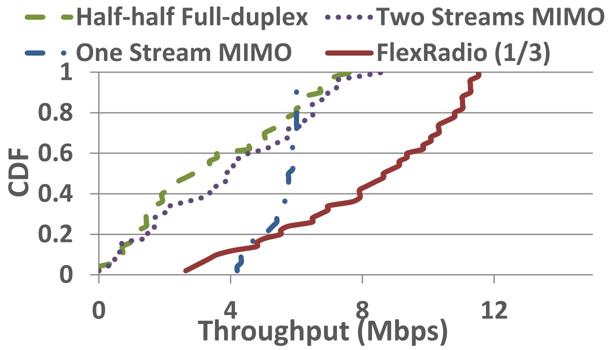


Figure 13: Throughput comparison between MIMO, half-half full-duplex and FlexRadio (1/3) configuration. One stream MIMO refers to the all TX chain in the MIMO transmitter transmit the same data while in the second setting, they are divided into two groups so that two streams are transmitted along the transmission.

Note that, under one stream MIMO, one node transmits the same data on all four RF chains and the other node receives on all of its four RF chains. While for 1/3 FlexRadio, only 3 RF chains are used for receiving by both the nodes. At very low SNR, the received SNR scales linearly with the number of receiver RF chains. This gives one stream MIMO a slight edge since it has one receive RF chain more than 1/3 FlexRadio. On average, FlexRadio provides a median gain of **1.51x over MIMO** and **2.85x over full-duplex.**

6 Related Work

Single RF-chain cancellation techniques. Prior RF cancellation techniques in existing full-duplex implementations [2, 3, 5–7, 9, 11–14] can be broadly classified into: *Passive (self interference suppression) and Active (Antenna cancellation, Analog cancellation)*. Passive suppression techniques provide electromagnetic isolation between the Tx and Rx antennas to minimize self interference, for instance, by using directional antennas, [9]. Active cancellation methods create a null at the receive antenna by sending an inverted copy of the transmitted signal, either over air (Antenna cancellation [6]) or through transmission line (Analog cancellation [12]). Antenna cancellation techniques typically require additional antennas (either for Tx, or Rx or both). FlexRadio's symmetrical RF cancellation design draws from these designs to reduce implementation complexity.

Multi-RF chain full-duplex systems. Recently, many researchers have demonstrated FD capability on multi-RF chains systems [2,3,8]. MIDU [2] employs two-level antenna cancellation. The authors propose a symmetric arrangement of Tx and Rx antennas such that the transmitted signals from a pair of TX antennas are offset by π at a given Rx as well as the received signals at a pair of Rx antennas from a given Tx antenna are offset by π. MIDU needs $2\times$ the number of antennas needed for a MIMO-FD node with the same number of RF chains.

Single-Antenna full-duplex systems. All the above implementations use at least one antenna for each active RF chain. In the case of antenna cancellation, or MIDU, multiple antennas are used *per* active RF chain. However, recent work [4] implements a full-duplex node (with one active TX RF chain and one active RX RF chain) using only a single antenna. This technique uses a circulator to provide isolation between the Tx and Rx paths. They achieve further cancellation using analog cancellation techniques implemented with passive delay lines and variable attenuators on the cancellation signal. The work in [3] extends this full-duplex design to MIMO radios. In [3], the authors implement a *six RF-chain full-duplex node (3 transmit RF-chains and 3 receive RF-chains)* using only 3 antennas. Since an equivalent 3 antenna MIMO node can activate *at most* 3 transmit or 3 receive RF chains, a 3 antenna MIMO node is essentially a 3 RF-chain MIMO node. While the full-duplex design in [3] almost doubles the capacity between two nodes over that of MIMO nodes with the same number of antennas, this comes at the cost of having *more active RF-chains*. On the other hand, FlexRadio exploits flexibility to realize a fundamental performance increase while *not using any additional active RF chains.*

7 Discussion and Conclusion

MAC layer Implications. The examples in Sec. 3 assumed the presence of a central node with knowledge of RF resource capabilities of the all the nodes in the network. Nodes can piggyback information of their RF resource capabilities (in terms of number of antennas, RF chains etc.) with packets exchanging channel state information. For example, in enterprise wireless networks APs can collect information from their respective clients and forward this information to a designated server over the backbone. The server can then determine optimum configuration for all the nodes in the network. Designing algorithms to exploit FlexRadio capability to maximize network performance is an open problem.

Extending beyond four RF chains. The four RF-chain FlexRadio prototype is applicable to many existing MIMO systems (the standard LTE system, for instance). However FlexRadio's design principles can extend to nodes with more than four RF chains. The extended design can leverage the geometrical symmetry of the symmetric antenna placement design to minimize cost, area and power consumption of the FlexRadio node.

The complexity reduction of the RF cancellation is based on the following observation: If multiple transmitter antennas are equidistant to a given receiver antenna, the cancellation signal of these TX chains can be *combined* before passing through a *single delay and attenuation block* to cancel out their *self interference* at the given receiver. If multiple sets of transmitters are equidistant, at different distances, to a given receiver, then the receiver needs an independent *delay and attenuation block* for each such set of transmitters. In general terms, for an N RF-chain FlexRadio system with a biased transmission order defined as: $N_1, N_2, \cdots, N_k$ are the K transmitters in K/N-K mode, the number of *delay and attenuation block*s is bounded by:

$\sum_{i=1}^{n-1}$ (distinct distances from set $\{N_{i+1}, N_{i+2}, \cdots N_n\}$ to N_i).

This governs the minimum complexity RF cancellation circuitry for our design. We present an antenna placement scheme for FlexRadio system with larger number of RF-chains by simply extending the antenna placement scheme of the four RF-chain FlexRadio node along all sides. For a FlexRadio system with more than four RF chains, additional antennas can be added to the four antenna arrangement using the following priority:

- On the centroids of the triangle formed by the excenter and the vertices of the four antenna arrangement on the side closest to the excenter.
- On the excenters along the three sides of the four antenna arrangement.
- The above steps extend the four antenna arrangement by making copies of its geometry along all its sides. The process can be repeated on the newly created copy until all the antennas corresponding to its respective RF-chains of the node are placed.

Conclusion. FlexRadio is a fundamentally new capability for a wireless node. By choosing the number of RF chains to transmit and receive, network-wide throughput gains are possible. These opportunities can be potentially recognized either centrally or in a distributed fashion. Further, the symmetric antenna placement design of FlexRadio ensures that realization of FlexRadio does not present a significant hardware overhead compared to a full-duplex design with same number of RF chains. Thus, we believe the performance gains of FlexRadio nodes are promising.

Acknowledgements

We would like to thank our shepherd, Lakshmi Subramanian, and the anonymous reviewers for their comments and suggestions to improve our paper. This work is supported by NSF CNS-1254032.

References

[1] ADIB, F., KUMAR, S., ARYAN, O., GOLLAKOTA, S., AND KATABI, D. Interference alignment by motion. In *MOBICOM* (2013).

[2] ARYAFAR, E., KHOJASTEPOUR, M. A., SUNDARESAN, K., RANGARAJAN, S., AND CHIANG, M. Midu: enabling mimo full duplex. In *MOBICOM* (2012).

[3] BHARADIA, D., AND KATTI, S. Full duplex mimo radios. In *NSDI* (2014).

[4] BHARADIA, D., MCMILIN, E., AND KATTI, S. Full duplex radios. In *SIGCOMM* (2013).

[5] BLISS, D., PARKER, P., AND MARGETTS, A. Simultaneous transmission and reception for improved wireless network performance. In *SSP* (2007).

[6] CHOI, J. I., JAIN, M., SRINIVASAN, K., LEVIS, P., AND KATTI, S. Achieving single channel, full duplex wireless communication. In *MOBICOM* (2010).

[7] DUARTE, M., AND SABHARWAL, A. Full-duplex wireless communications using off-the-shelf radios: Feasibility and first results. In *ASILOMAR* (2010).

[8] DUARTE, M., SABHARWAL, A., AGGARWAL, V., JANA, R., RAMAKRISHNAN, K. K., RICE, C. W., AND SHANKARANARAYANAN, N. K. Design and characterization of a full-duplex multi-antenna system for wifi networks. *CoRR abs/1210.1639* (2012).

[9] EVERETT, E., DUARTE, M., DICK, C., AND SABHARWAL, A. Empowering full-duplex wireless communication by exploiting directional diversity. In *ASILOMAR* (2011).

[10] GOLLAKOTA, S., PERLI, S. D., AND KATABI, D. Interference alignment and cancellation. *SIGCOMM* (2009).

[11] HONG, S. S., MEHLMAN, J., AND KATTI, S. R. Picasso: flexible rf and spectrum slicing. In *SIGCOMM* (2012).

[12] JAIN, M., CHOI, J. I., KIM, T., BHARADIA, D., SETH, S., SRINIVASAN, K., LEVIS, P., KATTI, S., AND SINHA, P. Practical, real-time, full duplex wireless. In *MOBICOM* (2011).

[13] KHOJASTEPOUR, M. A., SUNDARESAN, K., RANGARAJAN, S., ZHANG, X., AND BARGHI, S. The case for antenna cancellation for scalable full-duplex wireless communications. In *HoTNets-X* (2011).

[14] RADUNOVIC, B., GUNAWARDENA, D., KEY, P., PROUTIERE, R., SINGH, N., BALAN, V., AND DEJEAN, G. Rethinking indoor wireless mesh design: Low power, low frequency, full-duplex. In *WiMesh* (2010).

[15] RAHUL, H., HASSANIEH, H., AND KATABI, D. Sourcesync: a distributed wireless architecture for exploiting sender diversity. *SIGCOMM* (2011).

[16] SEN, S., CHOUDHURY, R. R., AND NELAKUDITI, S. Csma/cn: carrier sense multiple access with collision notification. *IEEE/ACM Transactions on Networking (TON) 20*, 2 (2012), 544–556.

[17] TSE, D., AND VISWANATH, P. *Fundamentals of Wireless Communication*. Cambridge University Press, 2005.

[18] XIE, X., AND ZHANG, X. Does full-duplex double the capacity of wireless networks? In *INFOCOM* (2014).

[19] YANG, Y., CHEN, B., SRINIVASAN, K., AND SHROFF, N. Characterizing the achievable throughput in wireless networks with two active rf chains. In *INFOCOM* (2014).

[20] YENAMANDRA, V., AND SRINIVASAN, K. Vidyut: exploiting power line infrastructure for enterprise wireless networks. In *SIGCOMM* (2014).

Towards WiFi Mobility without Fast Handover

Andrei Croitoru, Dragoș Niculescu, Costin Raiciu,
University Politehnica of Bucharest

Abstract

WiFi is emerging as the preferred connectivity solution for mobile clients because of its low power consumption and high capacity. Dense urban WiFi deployments cover entire central areas, but the one thing missing is a seamless mobile experience. Mobility in WiFi has traditionally pursued fast handover, but we argue that this is the wrong goal to begin with. Instead, we propose that mobile clients should connect to all the access points they see, and split traffic over them with the newly deployed MPTCP protocol. We let a mobile connect to multiple APs on the same channel, or on different channels, and show via detailed experiments and simulation that this solution greatly enhances capacity and reliability of TCP connections straight away for certain flavors of WiFi a/b/g. We also find there are situations where connecting to multiple APs severely decreases throughput, and propose a series of client-side changes that make this solution robust across a wide range of scenarios.

1 Introduction

Mobile traffic has been growing tremendously to the point where it places great stress on cellular network capacity, and offloading traffic to the ubiquitous WiFi deployments has long been touted as the solution. In dense urban areas, offloading between WiFi and cellular may not be needed at all: WiFi is always available because of uncoordinated deployments by many parties, and is preferable because it offers higher bandwidth and smaller energy consumption.

To improve the mobility experience, many solutions have been proposed that *coordinate* Access Points [1, 2]; however, real deployments are fragmented between multiple operators, which together cover entire central areas. This is the case, for instance, in pedestrian areas of the Bucharest city center with three different hotspot providers active. In this paper we address the problem of roaming through mixed, uncoordinated deployments of APs, without changes to the deployed infrastructure. We assume clients have access to multiple operators (perhaps due to roaming arrangements between operators), or to home-users' APs(also uncoordinated) as proposed by Fon[3]. Recent surveys [4, 5] show that in cities one can find tens of networks on most popular channels.

Traditional WiFi mobility techniques, as with all other L2 mobility mechanisms are based on the concept of fast handover: when a mobile client exits the coverage area of one Access Point (AP), it should very quickly find another AP to connect to, and quickly associate to it. There is a great wealth of research into optimizing fast handover including scanning in advance, re-using IP addresses to avoid DHCP, synchronizing APs via a backplane protocol, even the using additional cards[6] to reduce the association delay - see § 2 for more details. We think this is the wrong approach, for many reasons:

1. To start the handover mechanism, a client has to lose connectivity to the AP, or *break-before-make*
2. There is no standard way to decide which of the many APs to associate with for best performance
3. Once a decision is made, there is no way to dynamically adjust to changes in signal strength or load

We conjecture that the emerging standard of Multipath TCP (MPTCP) enables radical changes in how we use WiFi: use of multiple APs becomes natural, whether on the same channel or different ones, and the perennial handoff problem at layer 2 gets handled at layer 4 allowing for a clean, technology independent, end-to-end solution for mobility. In this paper we test the following hypothesis: *all WiFi clients should continuously connect to several access points in their vicinity for better throughput and connectivity.*

We carefully analyze the performance experienced by a mobile client locked into using a single WiFi channel and associating automatically to all the APs it sees, *without using any explicit layer 2 handover*. We run a mix of testbed experiments to test a few key usecases and simulations to analyze the generality of our testbed findings across a wide range of parameters and scenarios. We find that, surprisingly, the performance of this simple solution is very good out of the box for a wide range of scenarios and for many WiFi flavors (802.11a, b, g): a WiFi client connecting to all visible APs will get close to the maximum achievable throughput. We discuss in detail the reasons for this performance, namely the WiFi MAC behavior and its positive interaction with MPTCP. In particular, the *hidden terminal problem gets a constructive solution* with MPTCP, as subflows of a connection take turns on the medium instead of competing destructively.

We also find that performance is not always good for certain combination of standards (e.g. 802.11n and g), and for some rate control algorithms: in such cases the client downloads too much data from APs far away, reducing total throughput. To address these issues, we design, implement and test: (a) a novel client side estimation technique that allows the client to accurately estimate the efficiency of the downlink from an AP, and b) a principled client-side algorithm that uses ECN marking to help the MPTCP congestion controller to find the most efficient APs. Finally, we implemented an adaptive channel switch procedure that allows harvesting capacity from APs on different channels.

We ran several mobility experiments in our building, comparing our proposal to using regular TCP connected to the individual APs. The results show a truly mobile experience: our client's throughput closely tracks that of a TCP client connected to the best AP at any given location. We also show that striping traffic across APs naturally and fairly harvests bandwidth in contention situations with hidden and exposed terminals.

2 Background and Related Work

Fast Handover. The 802.11 standards were originally designed for uncoordinated deployments, and are not particularly amenable for high speed handovers. The handover is performed in three phases: scanning, authentication, and association. The first phase has been empirically shown [7] to be 95% of the handover time, and has been the target for most optimizations in the literature.

One approach to reduce the scanning delay is to probe for nearby APs before the mobile loses its current link. SyncScan [8] goes off channel periodically to perform the scan, so that the mobile has permanent knowledge about all APs on all channels. DeuceScan [9] is a prescan approach that uses a spatiotemporal graph to find the next AP. Mishra et al. [10] uses neighbor graphs to temporarily capture the topology around the mobile and speed up the context transfer between APs using IAPP. When additional hardware is available, [6] delegates the task of continuous probing to a second card.

For enterprise networks there are several opportunities to optimize the handover process. One is the use of a wireless controller that has a global view of all the APs and the associated mobiles in the entire network. This architecture is supported by all major vendors, because it offers many other advantages in addition to controlled association and guided handover. Another avenue for better handover is to eliminate it altogether, with the adoption of a single channel architecture [11] where multiple coordinated APs "pose" as a single AP by sharing the MAC, an architecture currently in use by Meru, Extricom and Ubiquiti. In these architectures, the wireless controller routes the traffic to and from the appropriate AP, so that a mobile never enters the handover process.

802.11r [12] optimizes the last two components of the handover ensuring that the authentication processes and encryption keys are established before a roam takes place. Authentication occurs only once, when a client enters the mobility domain. Subsequent roams within a mobility domain use cryptographic material derived from the initial authentication, decreasing roam times and reducing load on back-end authentication servers. 802.11k [13] provides for roaming assistance from the AP: when it perceives the mobile moving away, it sends a notification and possibly a site report with other AP options to maintain connectivity. Finally, a survey of handover schemes [14] mentions several other efforts in this direction and classifies them based on the incurred overhead, necessity of changes (mobile, AP, or both), compatibility with standards, and complexity.

All these layer 2 solutions do improve handover performance, but their availability depends on widespread support in APs and clients. Performing a handover is a decision that locks the client into one AP for a certain period of time, which leads to poor performance when truly mobile: there is no good way of predicting the throughput the client will obtain from one AP in the very near future. By using a transport layer mobility solution (see also [15, 16]), we sidestep the need to upgrade all AP and client hardware for mobility and, more importantly, allow a client to utilize multiple APs at the same time.

Channel Switch [17, 18, 19, 20] has been used to allow round robin access to APs on different frequencies. The client maintains association to each AP, and uses an IP address in that subnet. All these schemes rely on a client's powersave mode to have an AP buffer packets while the client is exchanging data with other APs on other channels. We also used this method in our implementation (see section 7). Spider [21] also investigates multiple associations on the same and on different channels, and concludes that for high speed, sticking to one channel yields better results.

AP selection is the problem of choosing the right AP based on signal strength, available wireless bandwidth, end-to-end bandwidth, RTT[22], current load. [23] estimates the wireless bandwidth that the client would receive by using timing of beacon frames. [24] shows that WLAN multi-homing is desirable from several angles: pricing, efficiency, fairness. [25] also uses a game theoretical approach to explore AP association strategies depending on delay probing. [26] proposes collaborative sharing of a fraction of an APs bandwidth, which enables better centralized load balancing. Divert [1] is a heuristic that selects the best AP for downlink and uplink, exploiting physical path diversity. Similarly, MRD [27] also exploits path diversity, but only for uplinks.

MPTCP is a recently standardized extension of TCP [28] that has been implemented by Apple in the IOS 7 mobile operating system; more mobile phone manufac-

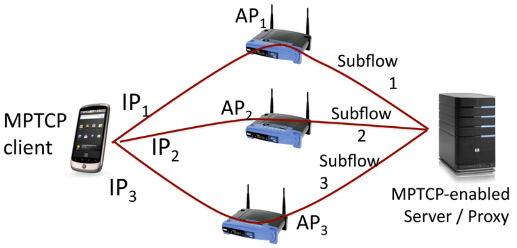


Figure 1: Instead of fast handovers, we propose that wireless clients associate to all the APs in range and use MPTCP to spread traffic across them.

turers are expected to follow suit.

The idea of associating to multiple APs for better robustness and throughput is not new [21, 29, 18]. What is missing is the ability of unmodified applications to benefit from multiple APs, as TCP uses a single interface by default. Multipath TCP enables unmodified apps to use these interfaces.

Our contribution in this paper is to study and optimize the interaction between the WiFi MAC and Multipath TCP. In contrast to previous works that focus mostly on channel switching, we examine carefully the case where the client is associated to multiple APs residing on the same channel. Our solution departs from the "one AP at a time" philosophy in existing works, allowing multiple APs to compete for the medium at packet level. Competition provides a number of features including elegantly solving hidden terminal issues, fast reaction to changes in channel conditions and throughput improvements even in certain static cases.

3 Towards an optimal solution for WiFi Mobility

Consider a wireless client that can associate to three distinct APs, as shown in Figure 1. Which one should the client pick and associate to? Prior work has shown that using signal strength is not a good indicator of future performance, so the client may actively probe or passively measure [23] all three APs briefly before deciding on picking one of them. However, this initially optimal choice may quickly become suboptimal because of multiple reasons outside the client's control: 1. the client may move; 2. other clients may use the medium, affecting this client's throughput and his choice; 3. the wireless channel to the chosen AP may have temporary short-term fluctuations, affecting its capacity (see evaluation in section 6 for an example).

The combination of these factors is impossible to predict in practice, and the best AP for any given client changes not only in mobility scenarios, but even when the client is stationary. All existing solutions that connect to a single AP are forced to be conservative, because fluctuations (flopping between APs) can affect performance; thus they all tend to stick to their choice for some time.

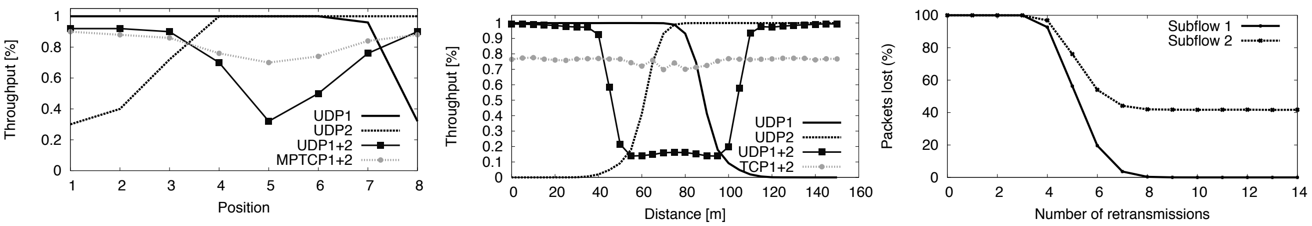
We observe that the emergence of MPTCP enables a radically different approach to WiFi mobility: instead of using only one AP at a time and doing handovers, **mobile clients should connect to all APs at any given time**. The solution is conceptually very simple, and is shown in Figure 1: we have the client associate to multiple APs, obtaining one IP address from each, and then rely on MPTCP to spread data across all the APs, with one subflow per AP. As the mobile moves, new subflows are added for new APs, while old ones expire as the mobile loses association to remote APs.

How should traffic be distributed over the different APs? As the client has a single wireless interface, it can only receive packets from one AP at a time, even if it is associated to multiple APs. Should the client spend an equal amount of time receiving data via each AP? This policy is optimal only when all APs offer equal throughput. In practice, one AP will offer the best performance, thus it is preferable for the client to transfer most data via this access point. However, all other feasible APs should be used to send probe traffic to ensure that the client can detect when conditions change and adapt quickly. While simple in principle, the key to this solution is understanding the interactions between MPTCP and the WiFi MAC. There are two high-level cases that need to be covered:

APs on the same wireless channel. There are three non-overlapping channels in 2.4GHz and more in 5GHz, but newer standards including 802.11n and 802.11ac allow bonding 2-8 of these 20Mhz channels to increase client capacity; the result is a smaller number of usable non-overlapping bonded channels (maximum three with 802.11ac, depending on the country).

If we disregard WiFi interference between APs, the theoretically optimal mobility solution is to always connect to every visible AP, and let MPTCP handle load balancing at the transport layer: if an AP has poor signal strength, its loss rate will be higher (because of lower bandwidth and similar RTTs) and the MPTCP congestion controller will simply migrate most of the traffic to the APs with better connectivity to the client. This way, handover delays are eliminated and the mobile enjoys continuous connectivity. But interference can be a major issue, and will be explored in depth in the next section.

APs on different wireless channels. In this case the mobile client must dynamically switch channels while associated to multiple APs, giving each AP the impression it is sleeping when in fact it is going on a different channel. Channel switching has already been proposed as a technique to aggregate AP backhaul capacity by a number of works including FatVAP [18] and Juggler[20]. We discuss the interactions between MPTCP and channel switching in section 7.



(a) Experimental results with 802.11a, 6Mbps: UDP flows systematically collide, while MPTCP subflows take turns on the medium.

(b) ns2 simulation of the same situation in 2a. In the middle region MPTCP exhibits higher variability because one subflow starves the other.

(c) The subflow sending packets infrequently experiences a much higher loss rate. This makes it hard for a (MP)TCP flow to escape its slowstart.

Figure 2: Hidden terminal (HT) experiments: using Multipath TCP results in very good throughput because one subflow monopolizes the air, while the other is starved.

4 Single-channel mobility

We implemented a prototype client that is locked on a single channel and continuously listens for beacons of known APs; when a new AP is found, the client creates a new virtual wireless interface[1] and associates to the AP, opening a new MPTCP subflow via the new AP. We ran this code on our 802.11a/b/g/n testbed without external interference, as well as in simulation to understand the interactions that can arise due to interference between different APs, and the extent to which this solution approximates the optimal one.

4.1 Hidden terminal experiments

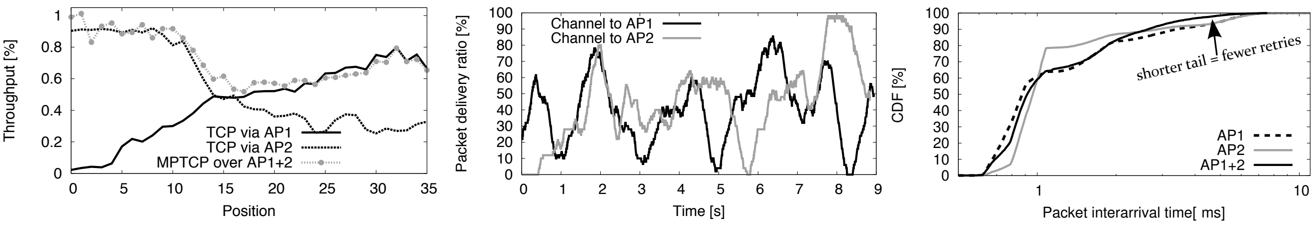
The first case we test is a pathological one: consider two APs that are outside CS range and the MPTCP client connects to both. Lack of CS means the CSMA mechanism does not function and the frames coming from the two APs will collide at the client. In fact, each AP is a hidden terminal for the other.

To run this experiment, we reduced the power of our two APs until they went out of CS, with the client still able to achieve full throughput to at least one AP at all test locations. Then, we place the client close to one AP and move it towards the other AP in discrete steps and measure the throughput for UDP and TCP via either AP (the status quo) as well as MPTCP. As shown in figure 2a, the graph exhibits three distinct areas. In the two areas close to either AP, neither UDP nor TCP throughput is affected: here the capture effect[30] of WiFi predominates, as packets from the closest AP are much stronger, and the effect of a collision is very small—the client will just decode the stronger packet as if no collision took place, and the subflow via the furthest AP will reduce its rate to almost zero because of repeated packet losses.

The area in the middle is more interesting. As we expected, the combined UDP throughput of two simultaneous iperf sessions is greatly diminished by the hidden terminal situation. However, by running two simultaneous MPTCP subflows, the combined throughput is surprisingly good. Repeated runs showed this result is robust, and we also confirmed this via ns2 simulation (Figure 2b). MPTCP connection statistics show that the high-level reason for the high throughput is that traffic is flowing entirely over one of the two subflows, while the other one is starved, experiencing repeated timeouts. This would suggest that the starved subflow is experiencing much higher loss rates, which would explain why it never gets off the ground properly.

To understand the reason of this behavior, we used simulation to measure the loss probability of the two subflows when contending for the medium. When subflow 1 is sending at full rate, subflow 2 sends a single packet which collides with a packet of subflow 1. The WiFi MACs will then backoff repeatedly until the max retransmission count has been reached, or until one or both packets are delivered. We run the simulation for a long time to experience many such episodes, and show the percentage of episodes each subflow experiences a loss in Fig.2c as a function of the retry count. When few retransmissions are allowed, both subflows lose a packet each when a collision happens, but the effect of the loss is dramatic for the second subflow pushing it to another timeout. As we allow more retransmissions, the loss probability is reduced dramatically: the second subflow loses around 40% of its packets if 6 or more retransmissions are allowed. The reason for the flattening of the line at 40% is the fact that the first sender always has packets to send, and when subflow 1 wins the contention for the first packet, its second packet will start fresh and again collide with the retransmissions from the second subflow, further reducing its delivery ratio. This also explains why subflow 1 experiences no losses after six retransmissions: either it wins the contention immediately, or it succeeds just after the second subflow delivers its

[1] Virtual interfaces are a standard Linux kernel feature: each interface has individual network settings (IP, netmask), MAC settings(association, retries), but share the PHY settings(channel, CS).



(a) Throughput of a client moving between AP_1 and AP_2: the MAC favors the sender with the better channel. Max throughput is 22Mbps.

(b) A fixed node has channels with a raw PDR≈50% to each AP. The two channels are weakly correlated.

(c) Packet inter-arrival rate exhibits a longer tail when a single AP is used. When both APs are sending, the tail is much shorter.

Figure 3: Carrier sense experiments: the client using MPTCP gets the throughput of the best TCP connection when close to either AP, and better throughput when in-between.

packet. In effect, we are witnessing a capture effect at the MPTCP level triggered by the interaction with the WiFi MAC. This behavior is ideal for the MPTCP client.

4.2 Carrier-sense experiments

The most common case is when a client connects to two APs on the same channel that are within carrier sense range of each other, so that the WiFi MAC will prevent both APs sending simultaneously. The mobile associates to both APs and again we move the client from one AP to the other in discrete steps. The performance of our MPTCP client in this case strongly depends on the rate control algorithm employed by the AP, and we explore a number of these to understand their effects.

First, we have our Linux APs use 802.11a and run the default Minstrel rate selection algorithm. The results are given in Fig. 3a, and they show that the throughput of the MPTCP client connected to both APs is as high as the maximum offered by any of the two APs. The reasons for this behavior are not obvious.

CASE I: In-between APs the client obtains slightly more throughput (10%) by using both APs than if we were using either AP in isolation. The fundamental reason lies at the physical layer: due to fading effects, individual channel quality varies quickly over time, despite both channels having a roughly equal longer-term capacity. The wide variability and burstiness of losses and successes in frame delivery is well documented in literature [31, 1]. To test this hypothesis, we simultaneously sent a low rate broadcast stream from each AP and measured their delivery ratios at the client. As broadcast packets are never retried, their delivery ratio captures the channel quality accurately; the low packet rate is used to ensure the two APs don't fight each other over the airtime, while still allowing us to probe both channels concurrently. The instantaneous packet delivery ratios computed with a moving window are shown in Fig. 3b, confirming that the two channels are largely independent.

The 802.11 MAC naturally exploits physical channel diversity: the sender that sees a better channel will decrease its contention window, and will be advantaged even more over the sender with a weaker channel. This behavior is experimentally verified by previous work [32] with several clients and bidirectional traffic to/from the APs. For our client downloading from two APs, when one has a slightly worse channel, it will lose a frame and double its contention window before retrying, leaving the other AP to better utilize the channel.

To validate our hypothesis, we analyzed inter-arrival times between packets for the client using either AP or both at the same time, and plotted the CDF in Figure 3c. The data shows that most packets arrive 1ms apart, and that AP1 prefers a higher PHY rate (24Mbps) while AP2 prefers a lower PHY rate (18Mbps) when used alone. Using both APs leads to inter-arrival times in between the two individual curves for most packets. The crucial difference is in the tail of the distribution, where using both APs results in fewer retries per packet. When one AP experiences repeated timeouts, the other AP will send a packet, thus reducing the tail.

The optimal AP changes at timescales of seconds, and a realistic way of harvesting this capacity is by connecting to multiple APs. Further experiments with 802.11n and simulations have shown this behavior is robust: even when the APs offer similar long-term throughput, a client connected to multiple APs will manage to harvest 10-20% more throughput, consistent with results in [1].

CASE II: One AP dominates. Consider now the cases when the client is closer to one AP; in such cases the most efficient use of airtime is to use *only* the AP that's closest to the client. In this case, the throughput of a client connected to all APs strongly depends on the rate selection algorithms used.

In the experiment in Fig.3a *minstrel* favors higher rates even at low signal strengths (with lower frame delivery rate), leading to more retries per packet for the far away AP. Each retry imposes a longer back-off time on the transmitter, allowing the AP with better signal strength to win the contention more often and thus to send more packets; this explains the near-optimal throughput enjoyed by the MPTCP client near either AP. This behavior is strictly enforced by the L2 conditions, and we verified that the choice of TCP congestion control has no effect on the distribution of packets over the two paths; the same results were obtained with UDP.

We also verified in the simulator that when two senders use the same rate, the MAC preference for the better sender holds regardless of the maximum number of retransmissions allowed (0 - 10). What happens when the AP farthest from the client sends using lower rates, thus reducing its frame loss rate? Simulations showed that the effect on total throughput can be drastic: the client connecting to both APs can receive less then half of the throughput it would get via the best AP. This is because lower rates give the farthest AP and the closest one similar loss rates and thus chances of winning the contention for the medium. However, packets sent at lower bitrate occupy more airtime, thus decreasing the throughput experienced by the client [33].

Is this case likely to appear in practice? We ran the same experiment on APs and clients running 802.11n in the 5GHz frequency band. When the client is close to one of the APs, the results differed from 802.11a/g: the throughput obtained with MPTCP was only approximately half the throughput obtainable via the closest AP.

Monitoring the PHY bitrates used by the transmitters shows that *minstrel_ht* (the rate control algorithm Linux uses for 802.11n) differs from *minstrel* significantly: instead of using aggressive bitrates and many retransmissions, *minstrel_ht* chooses the rates to ensure maximum delivery probability of packets. The block ACK feature of 802.11n is likely the main factor in this decision, as the loss notification now arrives at the sender after a whole block of frames has been transmitted (as much as 20): the sender can't afford to aggressively use high bitrates because feedback is scarce.

This issue is not limited to 802.11n: any rate control algorithm that offers packet-level fairness between multiple senders in CS range greatly harms the combined throughput achievable by MPTCP with multiple APs.

In summary, a client that associates to multiple APs and spreads traffic over them with MPTCP will receive close-to-optimal performance in a hidden-terminal situation, but in CS performance is strongly dependent on the rate adaptation algorithms employed by the APs, and these are outside the client's control.

5 Making MPTCP and the WiFi MAC play nicely together

There are two reasons for the suboptimal interaction between the 802.11 MAC and MPTCP: for one, the loss rate perceived by MPTCP on each subflow does not reflect the efficiency of the AP routing that subflow. In cases where packet-level fairness exists between APs, MPTCP sees the same loss rate on all subflows, and is unable to properly balance the load. Secondly, when subflows have shared bottlenecks, MPTCP assumes that sending traffic via the subflows will not affect the bottleneck capacity. This is not the case in single-channel WiFi setups, where sending traffic via a faraway AP will decrease the total throughput.

To fix these problems, it is simplest to stop using APs that decrease total throughput, but this comes at the expense of poorer performance when mobile. A more sensible option is **on-off**: receive data from a single AP at any point in time while all the others are shut-off, and periodically cycle through all available APs. **on-off** has already been proposed in the context of channel switching [18, 20, 34] and we use it in Section 7 to cope with APs on different channels. In our context, **on-off** can be implemented either by using the MP_BACKUP mechanism in MPTCP [28] which allows prioritizing subflows, or by relying on WiFi power save mode. It seems natural to extend the **on-off** solution for single channel APs as in [34], since there is no real cost of "switching" between APs on the same channel, beyond a couple of control messages: there is no wasted airtime. However, there are also a few fundamental drawbacks:

- Switching between APs increases packet delay and jitter, which affects interactive traffic. For instance, with a 200ms duty cycle, many packets experience RTTs that are 200ms longer that the path RTT.
- Gains from channel independence are lost.
- When multiple clients make independent decisions to switch between APs, they may end-up switching at the same time to the same AP, wasting capacity available elsewhere. Simulation results in §7 that show around 35% of the available capacity can be wasted in such cases.

Clients can monitor local PHY/MAC conditions accurately, but have a limited view of end-to-end downlink capacity available via an AP, because end-to-end loss rate and RTT are only available at the TCP sender. The sender, on the other hand, uses inaccurate metrics that are influenced by the WiFi behavior. For these reasons, our solution allows different APs to simultaneously send to the same client, while allowing the MPTCP congestion controller to direct traffic to the most efficient APs. In particular, our MPTCP client uses local WiFi information to find out which APs are preferable, and relays this information to the sender as additional loss notification. One non work-conserving way to relay this information is to have the client drop a percentage of packets. Instead, we use explicit congestion notification(ECN) to tell the server to avoid, if possible, the bad APs.

Our solution has two distinct parts discussed next: a novel client-side passive measurement technique that allows the client to accurately estimate the efficiency of each AP, and an algorithm that decides the amount of ECN marks that a client can set on a subflow to enable efficient airtime usage.

5.1 Measuring AP Efficiency

When deciding which AP it should prefer, the client needs to estimate the time T_i it takes on average for AP i to successfully send a packet, assuming the AP is alone in accessing the wireless medium. This metric purposely ignores the effect other WiFi clients may have on the transmission time by contending for the medium, or other clients that are serviced by the same AP. By comparing the resulting packet times, the client can decide which AP is preferable to use, and can signal the sender to steer traffic away from the other APs via ECN.

In contrast to previous work, we only care about the **hypothetical wireless bandwidth** from each AP to the client, as some of the interference from other APs is created by the client itself, so actual wireless bandwidth is not a good estimator.

We model the packet delivery time T (if the client were alone), when using the bitrate MCS at the AP and a PHY loss rate p with R retransmissions per packet, ignoring packets undelivered after R retries:

$$T=\sum_{i=0}^{R}[(\frac{MSS}{MCS}+K)\cdot(i+1)+C\cdot\sum_{j=0}^{i}\cdot 2^{j}]\cdot p^{i}\cdot(1-p) \quad (1)$$

In the model above, the first term measures the packet transmission time including the airtime used and K accounts for different WiFi constants such as SIFS, DIFS and the time needed to send the ACK at the lowest MCS (1Mbps). The term $C\cdot\ldots$ measures the time spent due to the contention interval, and models its increase on successive frame losses. Finally, the whole term is moderated by the probability that i retransmissions are needed to successfully send the packet.

The client knows the MCS used by the AP, however estimating the PHY loss rate is more difficult because it can only observe successful transmissions; for each successful transmission there may be an unknown number of retransmissions, which conceal the physical loss rate. Thus the obvious formula $delivery\_prob=\frac{N_{received}}{N_{total}}$ cannot be used at the client, as N_{total} is unknown.

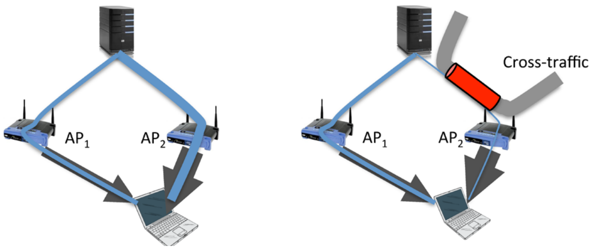


Figure 4: Scenario 1 (left): a client using AP_1 and AP_2 prefers AP_2 because of its more efficient use of airtime. Scenario 2 (right): moving all traffic to AP_2 with its better radio conditions is not the optimal strategy end-to-end

We avoid this problem by leveraging the "retry" bit present in the MAC header of every frame, signaling whether the frame is a retransmission. The client counts N_0, the number of frames received with the retry bit set to 0. All the other frames reaching the client will have the retry bit set to 1, and are counted as N_1. We recast the previous formula to measure the delivery probability only for frames that are delivered on the first attempt:

$$delivery\_prob=\frac{N_0}{N_0+N_1+N_{lost}} \quad (2)$$

The term N_{lost} captures packets that were not delivered by the AP despite successive attempts as shown by the sequence number present in the MAC header.

Implementation. To accurately estimate the delivery probability for all APs on a channel, the client maintains a FIFO queue of packets seen in a chosen interval, recording for each packet the time of arrival, its sequence number and its retry bit (10B in total). When new packets are received, or when timers expire, the packets outside the interval are purged, and N_0, N_1 and N_{lost} of the corresponding AP are modified accordingly. Our implementation uses an interval of 500ms, which results in an overhead per channel of around 10KB for 802.11a/g, and 100-200KB for 802.11n with four spatial streams.

5.2 Helping senders make the right choice

Consider the two scenarios depicted in Figure 4, where AP_2's packet time is shorter than AP_1's, and the two subflows going via AP_1 and AP_2 do not interfere at the last hop. MPTCP congestion control [35] requires that it does no worse than TCP on the best available path, and it efficiently uses network resources. MPTCP achieves the first goal by continuously estimating the throughput TCP would get on its paths using a simple model of TCP throughput, $B=\sqrt{\frac{2}{p}}\cdot\frac{MSS}{RTT}$. With this estimate, MPTCP adjusts its overall aggressiveness (total congestion window increase per RTT over all its subflows) so that it achieves at least the throughput of the best TCP. To achieve the second goal, MPTCP gives more of the increase to subflows with smaller loss rates.

In scenario 1, the throughput via AP_2 is higher than AP_1, resulting in a lower loss rate on the corresponding subflow and making the MPTCP sender send most of its traffic via AP_2. In scenario 2, other bottlenecks reduce the throughput available via AP_2, and the load balancing of traffic over paths will depend on the amount of loss experienced on AP_2. Either way, MPTCP will use its estimate of throughput available over AP_1 to ensure it achieves at least as much in aggregate.

Our target is to help MPTCP achieve the same goals when the two subflows via AP_1 and AP_2 interfere. For this to happen, we use ECN to signal the sender that AP_1 is worse and should be avoided when possible. Just making traffic move away from AP_1 is simple: the client will simply mark a large fraction of packets (e.g. 10%) received via AP_1 with the Congestion Experienced codepoint, which will duly push the sender to balance most of its traffic via AP_2. However, this approach will backfire in scenario 2, where MPTCP will stick to AP_2 and receive worse throughput.

To achieve the proper behavior in all these scenarios, the rate of ECN marks sent over AP_1 must be chosen carefully such that it does not affect MPTCP's estimation of TCP throughput via AP_1. Our goal is to ensure that the **MPTCP connection gets at least as much throughput as it would get via AP_1 if the latter is completely idle**. In particular, say the rate of ECN marks the client adds is δ. As the TCP congestion window depends on loss rate, the congestion window will decrease when we increase the loss rate. For the bandwidth to remain constant, we would like RTT_δ, the RTT after marking, to also decrease. In other words we would like for the following equation to hold:

$$B = \sqrt{\frac{2}{p}} \cdot \frac{MSS}{RTT} = \sqrt{\frac{2}{p+\delta}} \cdot \frac{MSS}{RTT_\delta} \qquad (3)$$

We assume the subflow via AP_1 is the unique subflow at that AP; congestion control at the sender will keep AP_1's buffer half-full on average. Thus, the average RTT observed by the client can be decomposed as $RTT = RTT_0 + \frac{BUF}{2} \cdot T_1$, where RTT_0 is the path RTT, and the second term accounts for buffering. Note that we use T_1, the average packet delivery time for AP_1 estimated by our metric. If our client reduced its RTT to RTT_0 by decreasing its congestion window, it would still be able to fill the pipe, and more importantly it would allow the sender to correctly estimate the bandwidth via AP_1. Using these observations and simplifying the terms, we rewrite the equation above as:

$$B = \sqrt{\frac{1}{p}} \cdot \frac{1}{RTT} = \sqrt{\frac{1}{p+\delta}} \cdot \frac{1}{RTT - T_1 \cdot \frac{BUF}{2}} \qquad (4)$$

Finally, knowing T_1 gives us an estimate of the maximum bandwidth $B = \frac{1}{T_1}$. We now have two equations with three unknowns: p, δ and BUF. Fortunately, we don't need to know the exact value of BUF; using a smaller value will only lead to a smaller value for δ, reducing our ability to steer traffic away from AP_1. To get an estimate of BUF, we note that nearly all wireless APs are buffered to offer 802.11a/g capacity (25Mbps) to single clients downloading from servers spread across the globe (i.e. an RTT of at least 100ms). This implies the smallest buffers should be around 2.5Mbit, which is about 200 packets of 1500 bytes. We use 200 as our estimate for BUF, and can now solve the two equations for δ. The closed form we arrive at is:

$$\delta = \frac{1}{2} \cdot \left(\frac{50 \cdot T_1^2}{RTT \cdot (RTT - 50 \cdot T_1)}\right)^2 \qquad (5)$$

δ depends on the interface (T_1) and the RTT of the subflow that will be marked, both of which are available at the client. Note that δ provides a maximum safe marking rate, and the actual marking rate used may be lower. For instance, marking rates in excess of 5% brings the TCP congestion window down to around 6 packets and makes TCP prone to timeouts.

In our implementation, the client computes the estimation of δ for every AP it is connected to. The client monitors the values of T_i for all of its interfaces, and sets ECN marks for subflows routed via interfaces with a packet time at least 20% larger than the best packet time across all interfaces. The 20% threshold is chosen to avoid penalizing good APs for short fluctuations in performance ECN marking happens before the packets get delivered to IP processing code at the client.

6 Evaluation

We have implemented our solution in the Linux 3.5.0 kernel, patched with MPTCP v0.89. Most of the changes are in the 802.11 part of the Linux kernel, and are independent of the actual NIC used. The patch has 1.3KLOC, and it includes code to compute the packet time for each wireless interface, the ECN marking algorithm, and channel switching support.

In this section we analyze each of our contributions in detail both experimentally and, where appropriate, in simulation. We first measure our client-side link estimation technique in a controlled environment. Next, we analyze the marking algorithm using 802.11n in the lab, and extensively in simulation to find it provides close to optimal throughput (90%) over a wide range of parameters. Next, we analyze fairness to existing clients and performance for short downloads. Finally, we run preliminary mobility tests "in-the-wild" using APs we do not control, finding that our solution does provide near-optimal throughput in real deployments.

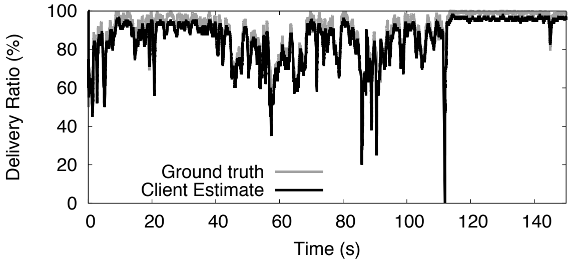


Figure 5: Client-side estimation of PHY delivery probability in 802.11a, fixed rate (54Mbps)

6.1 Client-side link quality estimation

To test the accuracy of our client-side estimation of PHY delivery probability, we setup one of our APs in 802.11a, place a sniffer within 1m of the AP, and place the client around 10m away from the AP. The AP's rate control algorithm is disabled, and we set the MCS to 54Mbps.

Both the client and the sniffer measure the average delivery ratio over a half-second window. The size of the window is a parameter of our algorithm: larger values take longer to adapt to changing channel conditions, while smaller values may result in very noisy estimations. Half a second provides a good tradeoff between noise and speed of adaptation. MPTCP congestion control (we use the OLIA algorithm [36]) reacts fairly quickly to moderate differences in loss rates (20% higher loss rate on one path). Experiments show that it takes between 1s and 2s for traffic to shift to the better path once an abrupt change has occurred, when the RTT is 50ms.

The client downloads a large file and we plot its estimation of the delivery probability (relation (2)) against the ground truth, as observed at the packet sniffer near the sender. Two minutes into the experiment we increase the transmit power of the AP to the max, thus improving the delivery probability. The results are given in Figure 5 and show that the client's estimation closely tracks the actual delivery ratio, and the mean error across the whole test is around 3%. We ran many experiments with 802.11g/n and observed similar behavior: client side estimation closely tracks the ground truth, and the mean error rate was under 5% in all our experiments.

Our metric is based on the assumption that the delivery ratio is independent of the state of the packet (the number of retries). This assumption is reasonable when packet losses occur due to channel conditions, but breaks down in hidden terminal situations, where a collision on the first packet will most likely trigger collisions on subsequent packets. In such cases, our metric's focus only on the initial transmissions will lead to errors, as follows:

- When competing traffic is sparse, our metric will overestimate the PHY delivery probability by around 10% in our tests.

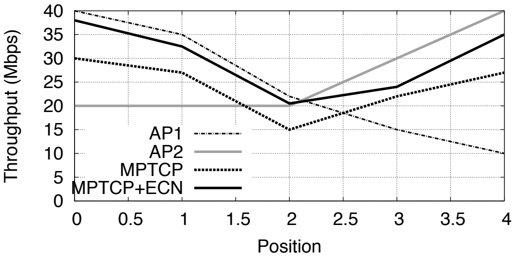


Figure 7: Throughput of a nomadic client at different position between AP1 and AP2 in 802.11n. MPTCP with ECN marking provides 85% of the optimal throughput.

- In heavy contention, one AP may be starved completely, and our client's estimate will be unreliable.

This drawback does not affect our overall solution: **we need client-side estimation only when the two APs are in carrier sense**. When in hidden terminal, our experiments show that the interaction between the MAC and Multipath TCP leads to a proper throughput allocation, and no further intervention via ECN is needed.

When a rate control algorithm picks a different rate to resend the same packet, that packet will not have its "retry" bit set despite being retransmitted. To understand whether this affects our results, we ran experiments as above but with rate control enabled; however the were no discernible differences in the efficacy of our algorithm.

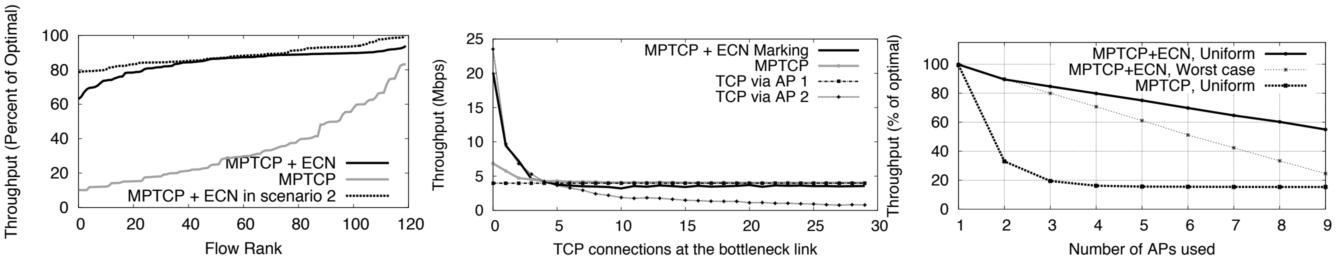
6.2 ECN Marking

We reran all the 802.11a/g experiments presented so far with our client-side load balancing algorithm on. We found that the marking did not influence the actual results: in particular, we verified that marking was not triggered in the channel diversity setup we discussed before.

For a static 802.11n client, we applied the ECN marking as indicated by relation (5). The results shown in Fig. 7 reveal that our metric and ECN algorithms work well together, pushing traffic away from the inefficient AP. Using the same setup, we then moved the client at walking speed between the APs, as the whole distance was covered in around 10s. The results (not shown) are much noisier, but show that the ECN mechanism still works fairly well overall; a similar result with a mix of 11n and 11g is later discussed in Figure 8. All further experiments presented in this paper are run with the ECN algorithm enabled, unless otherwise specified.

6.2.1 Simulation analysis

To understand how our proposed marking algorithm works in a wide range of scenarios, we also implemented it *htsim*, a scalable simulator that supports MPTCP and that has been used in previous works [37, 38]. *htsim*



(a) Flow throughput across a wide range of parameters. Marking achieves on average 85% of the optimal throughput.

(b) ECN Marking delivers near optimal throughput for scenario 2.

(c) Connecting to more APs reduces the total available throughput.

Figure 6: ECN simulation in *htsim*.

does not model 802.11; instead, we implemented a new type of queue that models the high level behavior of shared access in 802.11 DCF, namely: different senders' (AP) packets take different time for transmission on the medium, and when multiple senders have packets ready, packets are chosen according to a weighted fair-queueing algorithm, with predefined weights for the different APs.

Using this simple model, we can explore specified outcomes in the MAC contention algorithm, for example when AP1 wins contention four times more often than AP2, that are difficult to obtain in 802.11 setups simply by choosing different rate selection algorithm. Our simulated topology is shown in Fig. 4a, where the client is using both AP_1 and AP_2. In all our tests, AP_2 offers a perfect 802.11a/g connection (max 22Mbps), meaning that T_2, the packet time for AP_2 is set to 0.5ms.

We ran simulations testing all the combinations of parameters important in practice:: RTT (10, 25, 50 and 100ms), T_2 (from 1ms to 6ms), and the weights for different APs (1, 2, 4, 8 or 16). We ran a total of 120 simulations, and we present the throughput obtained as percentage of the optimal, sorting all values ascendingly. Figure 6a shows that the ECN marking algorithm is very robust: its average performance is 85% of the optimal (median is 87%), and its worst case is 65%. In contrast, MPTCP alone fares poorly: 34% throughput on average (28% in the median). Finally, the throughput of MPTCP in Scenario 2 is also robustly close to the optimal: average at 84% and median at 88%.

There are parameter combinations where the ECN algorithm is not as effective: when RTTs are small, δ is fairly high so ECN does manage to reduce the congestion window over AP_1. However, even a very small window of packets sent via AP_1 is enough to eat up a lot of airtime that would be better used by AP_2, and this effect is more pronounced when AP_1 wins the contention more often, because it has fewer retries.

In all experiments, we cap the marking rate to a maximum of 5% to avoid hurting throughput in Scenario 2. This is a direct tradeoff: the higher the allowed rate, the better MPTCP behaves in scenario 1, but the worse it behaves in scenario 2. The reason for this behavior is that the traditional formula used by MPTCP to estimate throughput over its subflows is overly optimistic for higher loss rates, where retransmit timeouts begin to dominate throughput.

To analyze scenario 2, we use a setup where $T_1 = 3ms$ (approx. 4Mbps), $RTT = 25ms$ and vary the number of TCP flows contending for the uplink of AP_2, whose speed is set to $25Mbps$. Figure 6b shows that MPTCP alone fails to deliver when the AP_2 uplink is idle, but obtains the maximum possible throughput when AP_2's uplink is busy (same as TCP over AP_2). MPTCP with ECN marking gets the best of both worlds: it closely tracks the performance of a single TCP flow via AP_2 when there is little contention for AP_2's uplink, and it stabilizes to just under 4Mbps when AP_2 uplink is congested.

Increasing the number of APs. We've looked at connecting to two APs until now. What happens when there are more APs the client can connect to? We ran an experiment where the best AP offers maximum rate, and we are adding a varying number of other APs. In our first experiment, we consider a worst case where all the added APs are poor: their packet times is set to $6ms$ (2Mbps); in our second experiment we distribute the packet times of the APs uniformly between $0.5ms$ and $6ms$, mimicking a more realistic situation, and plot both results in Figure 6c. The results show a linear drop in the throughput obtained as the number of APs increases when ECN is used, however the slope is steeper when all APs have poor performance. This graph shows that connecting to more than 3-4 APs, is a bad idea: the client should choose the few best APs and connect to those alone.

6.3 A mobility experiment

We now discuss a mobility experiment run in a building with mixed WiFi coverage: the user starts from a lab on the second floor of the building, goes down a flight of stairs and then walks over to the cafeteria. En route, the mobile is locked on channel 6 and can associate to five different APs, each covering different parts of the route. We repeat the experiment several times, each taking of around one minute, during which the client is ei-

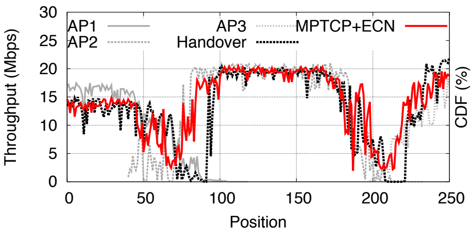


Figure 8: Mobility experiment: indoor client walking speed.

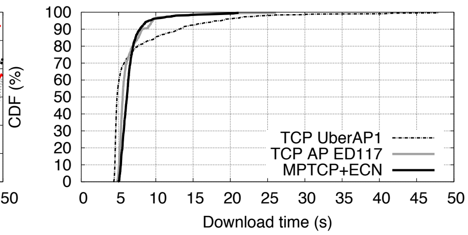


Figure 9: Static clients experience p formance variations outside their contr MPTCP offers predictable performance

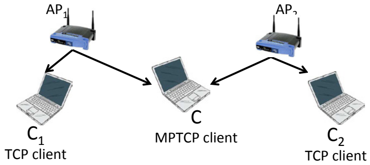


Figure 10: Experimental setup to test fairness

ther: a) using one AP at a time, with standard TCP; b) using MPTCP and associating to all APs it sees all the time; or c) performing handover between the different APs by using MPTCP. Our client monitors the beacons received over a 2s period, and switches to a new AP when it receives Δ more beacons than the current AP. It is well known that TCP performance suffers in cases of frequent mobility [39]. The same effect happens during MPTCP handovers, when a new subflow is created and has to do slowstart after switching to a new AP. In-between APs the client may flip-flop between APs based on its observed beacon count, reducing overall performance. To avoid this effect, we experimentally chose $\Delta = 10$.

In another experiment, the client slowly moves through the building at 1km/h, and the results are shown in Fig.8. At the beginning of the walk, the client has access to two Linux APs running minstrel, and these are also accessible briefly on the stairs, in the middle of the trace. The departments' deployment of uncoordinated APs (Fortinet FAP 220B) are available both in the lab at very low strength, on the stairs, and closer to the cafeteria. Our mobile client connects to at most three APs simultaneously. Throughout the walk, the throughput of our MPTCP mobile client closely tracks that of TCP running on the best AP in any given point; the handover solution suffers because it uses a $break-before-make$ strategy and throughput drops to nearly zero for 5-10s. We also noticed that in the beginning of the trace our ECN-based load balancing algorithm penalizes the subflow over the department AP—if we disable it, the throughput of MPTCP in that area drops dramatically.

6.4 Static clients

Our experiments so far show that connecting to multiple APs pays off when mobile. Is it worth doing the same when the client is static? We had our client connect to two APs (channel 11) visible in our lab and that we do not control, and the performance from both APs is similar. Our client repeatedly downloads the same 10MB file from one of our servers using either TCP over AP1, TCP over AP2 or MPTCP+ECN over both APs. We ran this experiment during 5 office hours, and present a CDF of the throughputs obtained in Figure 9. The figure shows there is a long tail in the throughput obtain via either AP because of external factors we do not control: other WiFi users, varying channel conditions, etc. The median download time for AP1 is 5s, 5.6s via AP2 and 6s with MPTCP (20% worse). However, MPTCP reduces the tail, cutting the 99% download time in half.

Power consumption While connected to different APs, the solution adds the following per AP costs: association and authentication handshakes, DHCP, and the null frames required whenever the mobile goes in and out of power save. These are negligible, as the bulk of the cost is due to the radio processing and the TCP/IP stack [40]. The energy cost of our solution is therefore dependent on the actual throughput achieved, which is near-optimal.

6.5 Fairness

We have focused our analysis so far on the performance experienced by a client connecting to multiple APs, and showed that there are significant benefits to be had from this approach. We haven't analyzed the impact this solution has on other clients: is it fair to them? Does our solution decrease the aggregate throughput of the system? In answering these questions, our goals are neither absolute fairness (WiFi is a completely distributed protocol and also inherently unfair), nor maximizing aggregate throughput (which may mean some distant clients are starved). Rather, we want our solution's impact on other clients to be *no worse than that of a TCP connection using the best available AP*.

The theory of MPTCP congestion control reassures us that two or more subflows sharing the same wireless medium will not get more throughput than a single TCP connection would. Also, if an AP has more customers, Multipath TCP will tend to steer traffic away from that AP because it sees a higher packet loss rate.

We used the setup shown in Figure 10 to test the behavior of our proposal. There are two APs each with a TCP client in their close vicinity, using 802.11a, and an MPTCP client C using both APs.

The first test we run has both APs using maximum power: when alone, all clients will achieve the maxi-

C conn. to	AP_1-C_1	AP_2-C_2	C
AP_1 (TCP)	5	10	5
AP_2 (TCP)	10	5	5
AP_1&AP_2	7	7	7

Table 1: APs&clients in close range: MPTCP provides perfect fairness (802.11a, throughput in Mbps).

C conn. to	AP_1-C_1	AP_2-C_2	C
AP_1(TCP)	3.5	13	3.5
AP_2(TCP)	10	5	5
AP_1&AP_2	10	5	5

Table 2: Client close to AP_2: MPTCP client behaves as TCP connected to AP_2

C conn. to	AP_1-C_1	AP_2-C_2	C
AP_1(TCP)	3.5	13.5	3
AP_2(TCP)	14	3	3
AP_1&AP_2	8.5	6.5	5

Table 3: Client in-between APs: MPTCP client improves overall fairness

mum rate in 802.11a, around 22Mbps. The results of the experiment are shown in table 1: when the client connects to both APs, the system achieves perfect fairness. In comparison, connecting to either AP alone will lead to an unfair bandwidth allocation. In this setup, MPTCP congestion control matters. If we use regular TCP congestion control on each subflow, the resulting setup is unfair: the MPTCP client receives 10Mbps while the two TCP clients each get 5Mbps.

We next study another instance of the setup in Fig. 10 where the APs, still in CS, are farther away, and while the TCP clients remain close to their respective APs, they get a lesser channel to the opposite AP. First, we place C close to AP_2. When C connects to AP_1, which is farther, it harms the throughput of C_1, and the overall fairness is greatly reduced. When C connects to both APs, its traffic flows via the better path through AP_2, offering optimal throughput without harming fairness (Table 2). When the client is between the two APs, traffic is split on both paths and the overall fairness is improved, while also increasing the throughput obtained by our client (Table 3).

In summary, by connecting to both APs at the same time and splitting the downlink traffic between them, MPTCP achieves better fairness in most cases, and never hurts traffic more than a TCP connection would when using the best AP.

7 Channel-switching

To connect to APs on different WiFi channels, clients can use channel switching, a technique supported by all NICs for probing. This technique, was proposed and shown to work in previous work [18, 19, 20, 21]; We implement a similar procedure, but with adaptation based on the actual bandwidth obtained on each channel.

Say the client spends a slot c_i on channel i, such that the sum of all slots equals the global duty cycle $C = \sum_i c_i$. While on channel i, the client measured the bandwidth it receives on that channel, b_i, by counting the number of bytes received in a slot and dividing it by c_i. We consider the following family of algorithms for channel switching:

$$c_i = \frac{b_i^\alpha}{\sum_j b_j^\alpha} \cdot C \tag{6}$$

The equation prescribes how to compute c_i for the next interval based on the throughput observed in the current interval, where the interval is a multiple of C. α dictates how aggressively we prefer the good channels over the bad ones: higher values lead to more time spent on the best channels. Choosing α strikes a tradeoff between throughput obtained and accurate probing that enables quick adaptation in channel conditions.

The discussion so far has assumed MPTCP is able to allow all APs on different channels to send at flat rate during their slot; in other words Multipath TCP manages to keep all the paths busy. Also note that there are no direct interactions between the MACs of the different APs during this time: enabling MPTCP to work over channel switching is a much easier task. All we need is to make sure the MPTCP subflows do not suffer frequent timeouts, which can occur due to:

- Wildly varying round-trip times leading to inaccurate values of the smoothed RTT estimator.
- Bursts of losses suffered when congestion windows are small and fast retransmit is not triggered.

The first problem is quite likely to appear during channel switching, as the senders will see bimodal RTT values: path RTT for packets sent during the channel's slot, and C for packets sent while outside the slot. To avoid this problem, we impose that C is smaller than the smallest possible RTO at clients, which must be higher than the delayed ACK threshold (200ms). Hence, our first restriction is that $C \leq 200ms$.

To avoid the second problem, we lower bound the time spent on any channel to a minimum value that allows the AP to send at least one packet per slot; this implies that the smallest slot has to be at least 10ms.

We have implemented channel switching support in the Linux kernel, together with the family of algorithms discussed above. With this implementation, we ran a series of experiments to understand the basic performance of channel switching in our context. We have the client associate to two APs in (channels 40 and 44, 802.11a) and modify the transmit power of the APs while we observe the adaptation algorithms at work. The results are shown in the table below. The experiments with both APs set at max power show that the channel switching overheads (of around 5ms in our measurements) reduce the total available throughput by around 10% when switching between two channels with a duty cycle of $200ms$. If we decrease the power of AP2, $\alpha = 2$ does a good job of increasing the slot of AP1, and obtaining 87% of the optimal throughput. In contrast, the algorithm $\alpha = 0$ assigns equal slot to both APs and throughput is the average of both APs' throughput:

Power for	TCP	TCP	MPTCP + switch	
AP1&AP2	AP1	AP2	$\alpha = 2$	$\alpha = 0$
Max & Max	20	20	18	18
Max & Low	20	14	17.5	16
Max & Low	20	5	17.5	12

The experiments show that MPTCP and channel switching play nicely together. We note that the experiments work similarly regardless the WiFi standard used. Our driver independent channel switching procedure, through its adaptive slot, makes it possible for an MPTCP based mobile to access capacity on independent channels in a fluid manner.

7.1 Channel switching with many users

The one key difference between the single channel and multi channel scenarios is the behavior when multiple users are connected to the same APs. When on the same channel, users tend to stick to the AP closest to them as our experiments showed in Section 4. When switching, the clients are not coordinated and will affect each other's throughput, depending on how their slots overlap. Intuitively, when multiple clients make independent switching decisions we expect the overall channel utilization to be suboptimal. We resort to simulation to understand these effects. We model a number of mobile clients connected to three APs on three distinct channels, and all clients can download from every AP at 22Mbps (i.e. the 802.11g perfect channels). The optimum is for the clients' speeds to sum up to 66Mbps. With channel switching, however, clients' slots will overlap and some channels will be idle while others may have two clients using them simultaneously. Our simulator uses a simplified model that assumes no channel switch overheads, and that bandwidth is shared equally amongst all clients using a channel. When computing slot times, we also add a number of ms chosen uniformly at random between 1-10ms, to model for random delays experienced by channel switching code due to interactions with the AP [18]. In the table below we plot the average throughput obtained as a percentage of the optimum.

Users	1	2	3	4	5	7	10
$\alpha = 2$	100	80	64	70	76	81	87

The results show worst results when three users contend for three channels and a third of the capacity is lost; if fewer or a lot more users share the channels, the effects are less pronounced. Note that these results also hold for the single channel setting, when the AP backhaul is the bottleneck (i.e. DSL scenarios).

8 Conclusion & Future work

We are witnessing a paradigm shift for wireless mobility: instead of using a single AP at any one time and racing to quickly change to the next when signal fades, the emerging MPTCP standard allows clients to use multiple APs in range and achieve a *truly mobile experience, independent of L2 handover*. Our main contribution in this paper is understanding the interaction between the WiFi MAC and MPTCP, and optimizing it for mobility.

Our experiments have shown that, in many cases, the load balancing job is done by the WiFi MAC (our carrier-sense experiments with 802.11a/g) or by interactions of the MAC and MPTCP congestion control (hidden terminal). However, there are situations when connecting to multiple APs can hurt throughput. We have offered a solution to these cases that utilizes a novel client-side measurement method together with an algorithm that uses ECN marking to enable the sender congestion controller to balance of traffic to the most efficient AP.

We have implemented and tested our solution both in simulation and in the Linux kernel. Nomadic and mobile experiments show our solution gets near-optimal throughput and is robust to changes in AP quality be they from client mobility or other factors.

In future work we plan to incorporate heuristics that allow WiFi clients to quickly associate to APs in vehicular mobility scenarios, as proposed by [41, 21]; a wider range of mobility experiments is also needed to ensure we have covered all the relevant situations. Additionally, we need to deploy our solution on mobile devices and quantify the energy consumption of our proposal, particularly in the channel switching scenario.

Our end-goal is to build a usable mobility solution which will combine channel switching as well as associating to multiple APs on the same channel. Understanding how our single channel solutions interact with switching is area worth of exploration.

Acknowledgements: This work was partly funded by Trilogy 2, a research project funded by the European Commission in its Seventh Framework program (FP7 317756), and by Mobil4, a research project funded by the Romanian government (UEFISCDI 11/2012). We also thank the anonymous reviewers and our shepherd Venkat Padmanabhan for their insightful feedback.

References

[1] A. Miu, G. Tan, H. Balakrishnan, and J. Apostolopoulos, "Divert: fine-grained path selection for wireless lans," in *Proceedings of the 2nd international conference on Mobile systems, applications, and services*, pp. 203–216, ACM, 2004.

[2] R. Murty, J. Padhye, R. Chandra, A. Wolman, and B. Zill, "Designing high performance enterprise Wi-Fi networks.," in *NSDI*, vol. 8, pp. 73–88, 2008.

[3] B. T. F. Network, "http://www.btfon.com/."

[4] A. Farshad, M. K. Marina, and F. Garcia, "Urban wifi characterization via mobile crowdsensing," in *Network Operations and Management Symposium (NOMS), 2014 IEEE*, pp. 1–9, IEEE, 2014.

[5] A. S. Engineering and Q. Associates, "Study on the use of wi-fi for metropolitan area applications." Final Report for Ofcom, April 2013.

[6] V. Brik, A. Mishra, and S. Banerjee, "Eliminating handoff latencies in 802.11 WLANs using multiple radios: Applications, experience, and evaluation," in *Proceedings of the 5th ACM SIGCOMM Conference on Internet Measurement*, IMC '05, (Berkeley, CA, USA), pp. 27–27, USENIX Association, 2005.

[7] A. Mishra, M. Shin, and W. Arbaugh, "An empirical analysis of the IEEE 802.11 MAC layer handoff process," *SIGCOMM Comput. Commun. Rev.*, vol. 33, April 2003.

[8] I. Ramani and S. Savage, "SyncScan: practical fast handoff for 802.11 infrastructure networks," in *INFOCOM 2005. 24th Annual Joint Conference of the IEEE Computer and Communications Societies. Proceedings IEEE*, vol. 1, pp. 675–684, IEEE, 2005.

[9] Y.-S. Chen, M.-C. Chuang, and C.-K. Chen, "Deucescan: Deuce-based fast handoff scheme in IEEE 802.11 wireless networks," *Vehicular Technology, IEEE Transactions on*, vol. 57, pp. 1126–1141, March 2008.

[10] M. Shin, A. Mishra, and W. A. Arbaugh, "Improving the latency of 802.11 hand-offs using neighbor graphs," in *Proceedings of the 2Nd International Conference on Mobile Systems, Applications, and Services*, MobiSys '04, (New York, NY, USA), pp. 70–83, ACM, 2004.

[11] "Virtual cells: The only scalable multi-channel deployment." MERU white paper, 2008.

[12] "IEEE standard for information technology– local and metropolitan area networks– specific requirements– part 11: Wireless LAN medium access control (MAC) and physical layer (PHY) specifications amendment 2: Fast basic service set (BSS) transition," *IEEE Std 802.11r-2008 (Amendment to IEEE Std 802.11-2007 as amended by IEEE Std 802.11k-2008)*, pp. 1–126, July 2008.

[13] "IEEE standard for information technology– local and metropolitan area networks– specific requirements– part 11: Wireless LAN medium access control (MAC) and physical layer (PHY) specifications amendment 1: Radio resource measurement of wireless LANs," *IEEE Std 802.11k-2008 (Amendment to IEEE Std 802.11-2007)*, pp. 1–244, June 2008.

[14] S. Pack, J. Choi, T. Kwon, and Y. Choi, "Fast-handoff support in IEEE 802.11 wireless networks," *IEEE Communications Surveys and Tutorials*, vol. 9, no. 1-4, pp. 2–12, 2007.

[15] A. C. Snoeren and H. Balakrishnan, "An end-to-end approach to host mobility," in *Proc. Mobicom*, ACM, 2000.

[16] W. M. Eddy, "At what layer does mobility belong?," *Communications Magazine, IEEE*, vol. 42, no. 10, pp. 155–159, 2004.

[17] R. Chandra and P. Bahl, "Multinet: Connecting to multiple IEEE 802.11 networks using a single wireless card," in *INFOCOM 2004. Twenty-third AnnualJoint Conference of the IEEE Computer and Communications Societies*, vol. 2, pp. 882–893, IEEE, 2004.

[18] S. Kandula, K. C.-J. Lin, T. Badirkhanli, and D. Katabi, "FatVAP: aggregating AP backhaul capacity to maximize throughput," in *Proceedings of the 5th USENIX Symposium on Networked Systems Design and Implementation*, NSDI'08, (Berkeley, CA, USA), pp. 89–104, USENIX Association, 2008.

[19] D. Giustiniano, E. Goma, A. Lopez, and P. Rodriguez, "WiSwitcher: an efficient client for managing multiple APs," in *Proceedings of the 2nd ACM SIGCOMM workshop on Programmable routers for extensible services of tomorrow*, pp. 43–48, ACM, 2009.

[20] A. J. Nicholson, S. Wolchok, and B. D. Noble, "Juggler: Virtual networks for fun and profit," *IEEE Transactions on Mobile Computing*, vol. 9, pp. 31–43, Jan. 2010.

[21] H. Soroush, P. Gilbert, N. Banerjee, B. N. Levine, M. Corner, and L. Cox, "Concurrent wi-fi for mobile users: Analysis and measurements," in *Proceedings of the Seventh COnference on Emerging Networking EXperiments and Technologies*, CoNEXT '11, (New York, NY, USA), pp. 4:1–4:12, ACM, 2011.

[22] A. J. Nicholson, Y. Chawathe, M. Y. Chen, B. D. Noble, and D. Wetherall, "Improved access point selection," in *Proceedings of the 4th International Conference on Mobile Systems, Applications and Services*, MobiSys '06, (New York, NY, USA), pp. 233–245, ACM, 2006.

[23] S. Vasudevan, K. Papagiannaki, C. Diot, J. Kurose, and D. Towsley, "Facilitating access point selection in IEEE 802.11 wireless networks," in *Proceedings of the 5th ACM SIGCOMM conference on Internet Measurements*, pp. 26–26, USENIX Association, 2005.

[24] S. Shakkottai, E. Altman, and A. Kumar, "Multihoming of users to access points in WLANs: A population game perspective," *Selected Areas in Communications, IEEE Journal on*, vol. 25, no. 6, pp. 1207–1215, 2007.

[25] B. alexander Cassell, T. Alperovich, M. P. Wellman, and B. Noble, "Access point selection under emerging wireless technologies," 2011.

[26] O. B. Karimi, J. Liu, and J. Rexford, "Optimal collaborative access point association in wireless networks," in *Proc. IEEE INFOCOM 2014*, 2014.

[27] A. Miu, H. Balakrishnan, and C. E. Koksal, "Improving loss resilience with multi-radio diversity in wireless networks," in *Proceedings of the 11th annual international conference on Mobile computing and networking*, pp. 16–30, ACM, 2005.

[28] A. Ford, C. Raiciu, M. Handley, and O. Bonaventure, "TCP Extensions for Multipath Operation with Multiple Addresses," rfc6824, IETF, 2013.

[29] A. Balasubramanian, R. Mahajan, A. Venkataramani, B. N. Levine, and J. Zahorjan, "Interactive wifi connectivity for moving vehicles," *ACM SIGCOMM Computer Communication Review*, vol. 38, no. 4, pp. 427–438, 2008.

[30] A. Kochut, A. Vasan, A. U. Shankar, and A. Agrawala, "Sniffing out the correct physical layer capture model in 802.11 b," in *Network Protocols, 2004. ICNP 2004. Proceedings of the 12th IEEE International Conference on*, pp. 252–261, IEEE, 2004.

[31] D. Aguayo, J. Bicket, S. Biswas, G. Judd, and R. Morris, "Link-level measurements from an 802.11 b mesh network," in *ACM SIGCOMM Computer Communication Review*, vol. 34, pp. 121–132, ACM, 2004.

[32] S. Choi, K. Park, and C.-k. Kim, "On the performance characteristics of WLANs: Revisited," *SIGMETRICS Perform. Eval. Rev.*, vol. 33, pp. 97–108, June 2005.

[33] M. Heusse, F. Rousseau, G. Berger-Sabbatel, and A. Duda, "Performance anomaly of 802.11b," in *INFOCOM 2003. Twenty-Second Annual Joint Conference of the IEEE Computer and Communications. IEEE Societies*, vol. 2, pp. 836–843, IEEE, 2003.

[34] D. Giustiniano, E. Goma, A. Lopez Toledo, I. Dangerfield, J. Morillo, and P. Rodriguez, "Fair wlan backhaul aggregation," in *Proceedings of the Sixteenth Annual International Conference on Mobile Computing and Networking*, MobiCom '10, (New York, NY, USA), pp. 269–280, ACM, 2010.

[35] D. Wischik, C. Raiciu, A. Greenhalgh, and M. Handley, "Design, implementation and evaluation of congestion control for multipath tcp," in *Proceedings of the 8th USENIX conference on Networked systems design and implementation*, NSDI'11, (Berkeley, CA, USA), pp. 8–8, USENIX Association, 2011.

[36] R. Khalili, N. Gast, M. Popovic, U. Upadhyay, and J.-Y. Le Boudec, "Mptcp is not pareto-optimal: Performance issues and a possible solution," in *Proceedings of the 8th International Conference on Emerging Networking Experiments and Technologies*, CoNEXT '12, (New York, NY, USA), pp. 1–12, ACM, 2012.

[37] D. Wischik, C. Raiciu, A. Greenhalgh, and M. Handley, "Design, implementation and evaluation of congestion control for multipath TCP," in *Proc. Usenix NSDI 2011*.

[38] C. Raiciu, S. Barre, C. Pluntke, A. Greenhalgh, D. Wischik, and M. Handley, "Improving datacenter performance and robustness with multipath tcp," in *Proceedings of the ACM SIGCOMM 2011 Conference*, SIGCOMM '11, (New York, NY, USA), pp. 266–277, ACM, 2011.

[39] P. Manzoni, D. Ghosal, and G. Serazzi, "A simulation study of the impact of mobility on tcp/ip," in *Network Protocols, 1994. Proceedings., 1994 International Conference on*, pp. 196–203, Oct 1994.

[40] A. Garcia-Saavedra, P. Serrano, A. Banchs, and G. Bianchi, "Energy consumption anatomy of 802.11 devices and its implication on modeling and design," in *Proceedings of the 8th International Conference on Emerging Networking Experiments*

and Technologies, CoNEXT '12, (New York, NY, USA), pp. 169–180, ACM, 2012.

[41] J. Eriksson, H. Balakrishnan, and S. Madden, "Cabernet: vehicular content delivery using wifi," in *Proceedings of the 14th ACM international conference on Mobile computing and networking*, pp. 199–210, ACM, 2008.

Securing RFIDs by Randomizing the Modulation and Channel

Haitham Hassanieh MIT
Jue Wang MIT
Dina Katabi MIT
Tadayoshi Kohno University of Washington

Abstract– RFID cards are widely used in sensitive applications such as access control and payment systems. Past work shows that an eavesdropper snooping on the communication between a card and its legitimate reader can break their cryptographic protocol and obtain their secret keys. One solution to this problem is to install stronger encryption on the cards. However, RFIDs' size, power, and cost limitations do not allow for strong encryption protocols. Further, changing the encryption on the cards requires revoking billions of cards in consumers' hands, which is impracticable.

This paper presents RF-Cloak, a solution that protects RFIDs from the above attacks, without any changes to today's cards. RF-Cloak achieves this performance using a novel transmission system that randomizes both the modulation and the wireless channels. It is the first system that defends RFIDs against MIMO eavesdroppers, even when the RFID reader has no MIMO capability. A prototype of our design built using software radios demonstrates its ability to protect commercial RFIDs from both single-antenna and MIMO eavesdroppers.

1. Introduction

Ultra-low power RFIDs are widely used in a variety of sensitive applications such as access control, payment systems, and asset tracking [21, 51, 71]. Some of the most well-known examples include the U.S. Passport Card, Zipcar key, MasterCard PayPass, RFID-equipped pharmaceuticals, and MBTA subway cards [38, 44, 45, 55, 72]. As a result of their ultra-low cost, ultra-low power requirements, these systems typically adopt weak encryption protocols [34, 59] or lack encryption altogether [63], leaving them widely exposed to security threats [38, 49].

Past attacks on commercial RFID systems have employed passive eavesdropping [10, 22, 58, 67]. In these attacks, an adversary snooping on the wireless medium intercepts the conversation between a legitimate RFID reader and an RFID card to obtain the sensitive data transmitted by the card. For example, the secret key in over one billion MIFARE Classic cards, widely used in access control and ticketing systems, can be obtained in real-time from an overheard conversation [22]. Similarly, the cipher used in RFID-based anti-theft devices for modern cars has been broken in under 6 minutes using eavesdropped information [67].

In theory, eavesdropping attacks can be addressed with more sophisticated encryption protocols than those typically used in RFIDs. Such an approach, however, would translate into more expensive, power-consuming cards, which goes against the main goal of the RFID industry, namely to dramatically reduce the size and cost of RFIDs so as to allow ubiquitous use [21]. Further, replacing the encryption requires revoking billions of RFIDs in consumers' hands, an impractical and costly endeavor.

In this paper, we introduce RF-Cloak, a system that defends RFIDs against eavesdroppers, without requiring any modifications to the RFID cards. RF-Cloak exploits that RFID cards do not generate their own transmission signal; they communicate by reflecting the signal transmitted by the RFID reader. In today's RFID systems, the reader transmits a constant waveform $c(t)$, and a nearby card *multiplies* (i.e., modulates) this waveform by its data x through reflection, producing $x \cdot c(t)$. In RF-Cloak, we replace the reader's constant waveform, $c(t)$, by a random signal, $r(t)$, which also makes the card's reflected message, $x \cdot r(t)$, appear random. Since the eavesdropper does not know the random waveform, he cannot extract the card's data from what he hears. In contrast, the reader is the one who generates the random waveform, and thus is able to decode by removing its effect. We refer to this technique as random modulation. We formally analyze it and characterize its security guarantees.

Random modulation is effective at defending against a single-antenna eavesdropper. However, random modulation alone cannot defend against a more powerful MIMO eavesdropper. This vulnerability is due to the fact that a MIMO system with n receivers can separate n signals [36, 65], which allows a MIMO eavesdropper to separate the card's signal from that of the reader. This is a fundamental problem with defending against MIMO eavesdroppers. The solution to this problem is to use a MIMO system on the reader that has at least as many transmitters as there are receivers on the MIMO eavesdropper [36]. Such a solution, however, creates a MIMO battle between the reader and the eavesdropper, where the reader has to keep increasing its MIMO transmitters to match the eavesdropper's MIMO capability.

In RF-Cloak, we present a novel solution that enables a reader with no MIMO capability to securely communicate with insecure cards, even in the presence of MIMO eavesdroppers. Specifically, a MIMO system relies on the channels being relatively static within a packet to be able to decode. Our key idea is to randomize the wireless channels from the reader to the MIMO eavesdropper to prevent it from correctly decoding. We analyze the impact of channel randomization and prove that it enables a reader with no MIMO capability to overcome a MIMO eavesdropper, even if it has a very large MIMO system.

To implement channel randomization in practice, we leverage recent results in wireless communication which show that, due to multipath, even small motion of the antenna can create large variations in the wireless channel [2, 50]. Thus, our system uses a rotating frame with multiple antennas, and randomly switches between the antennas using rapid switches.[1] We empirically show that this creates fast varying channels with a random distribution. We note that our design uses a single transmit chain on the reader –i.e., no MIMO. However, it provides the channel diversity of a MIMO transmitter with a huge number of antennas, which renders a MIMO eavesdropper unable to decode.

We study RF-Cloak's security guarantees both analytically and empirically. In particular, we implement the RF-Cloak reader on USRP software radios and evaluate it with commercial RFIDs in both the HF and UHF bands. Our evaluation reveals the following:

- Random modulation is effective at protecting RFIDs from single-antenna eavesdroppers. When the eavesdropper uses the optimal decoder which is the maximum likelihood decoder, he experiences a mean bit error rate of 49.8% for HF RFIDs and 50.3% for UHF RFIDs (and a standard deviation of 0.8% for HF and 2.3% for UHF), which is similar to the bit error rate of a random guess. On the other hand, the trusted RF-Cloak reader continues to be able to decode the RFID message.
- Combining random modulation with channel randomization, an RF-Cloak reader with no MIMO capability causes the mean bit error rate of a MIMO eavesdropper to be 50%, even if the eavesdropper has a MIMO system with 3, 4 or 5 receivers. The standard deviation ranges between 1.2% and 2.9%, depending on the number of receivers at the eavesdropper. Hence, RF-Cloak provides an effective mechanism to defend against a MIMO eavesdropper.

Contributions: This paper presents the first system that protects unmodified RFID cards from eavesdropping attacks, even if the eavesdropper has a large MIMO system and the reader has no MIMO capability. The paper introduces novel algorithms that randomize both the modulation and the wireless channels to the eavesdropper. It analytically proves its security guarantees and empirically demonstrates the benefits of its design. We believe that RF-Cloak addresses a real world problem that threatens the security of commercial RFIDs such as those used in car anti-theft solutions [67], and MBTA subway payment control [22].

2. Threat Model

We address passive eavesdropping attacks on commercial RFID cards in the HF and UHF bands, including cards with and without cryptographic protection.[2] In this attack, an adversary listening on the wireless medium intercepts the conversation between a legitimate reader and an RFID card and seeks to obtain confidential information contained in the card. In the simplest case, the adversary can learn the ID of the card, which threatens the privacy of the party carrying the card and enables cloning attacks. The adversary may also obtain sensitive data transmitted by the card, such as biometric information and passwords. Further, the adversary can reverse engineer the encryption and extract the secret key based on the eavesdropped information [10, 22].

The adversary may use standard or custom-built hardware with high receiver sensitivity including multi-antenna MIMO devices. Also, he may be in any location with respect to the card and the reader.

We secure the communication from the RFID card to the reader. We assume the commands transmitted from the reader to the card do not contain sensitive information. This assumption is justified since for HF cards (e.g., MIFARE), listening to the reader's messages alone does not allow the eavesdropper to extract the secret key and decode the card's encrypted data [10, 22]. For UHF cards, this assumption is satisfied as long as the reader acknowledges cards using only their temporary IDs, an option readily available for today's RFID readers [19].

We also assume that the reflected signal from the RFID card is significantly weaker than the direct signal from the reader. This assumption is valid for both HF and UHF systems [7, 18, 53]. In practice, the reflection is 20 to 30 dB weaker than the direct high power RF signal generated by the reader, because the card's circuit reflects only a small portion of the power it receives [37, 56].

Finally, this paper focuses on passive attacks as opposed to active attacks, in which an adversary repeatedly queries an RFID card to infer the secret key or obtain confidential information. Active RFID attacks are harder to mount than passive attacks. First, they have a shorter range because the attacker needs to power the RFID card [28, 29, 38, 49]. For example, for HF RFIDs,

[1] Cheap switches [20] can switch every few microseconds, which is faster than individual bits in an RFID transmission.

[2] Eavesdropping attacks have been successfully mounted on a variety of RFIDs that employ cryptographic protection [10, 22].

an active adversary needs to be within a few centimeters from the card whereas a passive eavesdropper can be more than 4 meters away [28]. Second, there are few practical and commercial solutions for protecting RFIDs from active attacks, including shielding sleeves which are used in US Passport Cards [38], RFID blocking wallets [52, 64], RFID reader detectors [43].

3. RFID Communication Primer

RFIDs mainly operate in two frequency bands: the High Frequency band (HF 13.56 MHz), where the communication range is about 10 cm [7], and the Ultra High Frequency band (UHF 902 MHz–928 MHz), where the communication range can reach a few meters [11]. RF-Cloak protects both types of RFIDs from eavesdropping attacks.

RFID cards do not generate their own transmission signals. Instead, they are powered and activated by the waveform coming from the RFID reader, through inductive coupling in the HF band [7] or RF backscatter communication in the UHF band [11]. In both UHF and HF systems, the reader continuously transmits a high power RF signal $c(t)$, and a nearby RFID card conveys its message by switching on and off its reflection of the reader's signal through a mechanism called load modulation. In particular, when the card switches on its load to reflect the reader's signal, its signal on the air appears as $x_1 \cdot c(t)$, where x_1 represents the fraction of the reader's signal reflected by the card. When the card switches off its reflection via open circuit, its signal on the air appears as $x_0 \cdot c(t)$, where x_0 is almost 0 and $x_0 \ll x_1 \ll 1$.

In current RFID systems, during the card's reply, the reader's baseband signal is a constant waveform $c(t) = A$, where A is a constant complex value. A nearby wireless receiver receives a weighted sum of the reader's signal and the reflected signal from the card:

$$y(t) = h_{reader \to receiver} \cdot c(t) + h_{card \to receiver} \cdot x(t) \cdot c(t) \quad (1)$$

$x(t)$ is the card's data message, $h_{reader \to receiver}$ is the wireless channel from the reader to the receiver, and $h_{card \to receiver}$ represents the channel of the card's reflected signal at the receiver i.e., it is a combination of the channel from the reader to the card with the channel from the card to the receiver. Note that the receiver in the above equation can be the reader itself or an eavesdropper.

4. RF-Cloak: Randomized Modulation

We first describe RF-Cloak's random modulation scheme, which protects RFIDs from single-antenna eavesdroppers.

In RFID systems, the reader transmits a query command and a nearby RFID card replies to it with its data. During the card's reply, the reader needs to continue transmitting a high power RF signal on which the card modulates its data, as detailed in §3. RF-Cloak randomizes this modulation of the card's data. To do so, instead of transmitting a constant signal as in today's RFID systems, an RF-Cloak reader transmits a random signal $r(t)$ during the card's reply.

Here we focus on two design goals. First, we ensure that an adversary cannot predict or learn the random modulation $r(t)$ to decode the card's data. Second, the RF-Cloak reader needs to decode with an accuracy comparable to the case where a reader uses a constant waveform to read the card.

4.1 Ensuring the Eavesdropper Cannot Decode

Recall from §3 that the eavesdropper's receives:

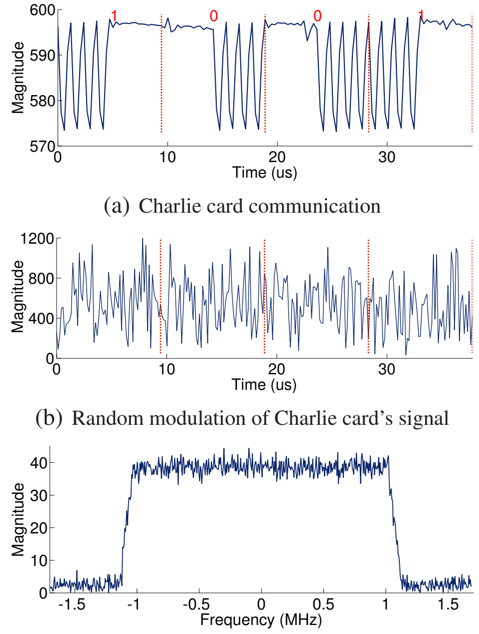
$$y(t) = h_{reader \to eve} \cdot r(t) + h_{card \to eve} \cdot x(t) \cdot r(t), \quad (2)$$

where $r(t)$ is the reader's random signal, $x(t)$ is the card's signal, and $h_{reader \to eve}$ and $h_{card \to eve}$ are the direct and reflected channels from the reader and the card respectively. To ensure the eavesdropper cannot decode, $r(t)$ should hide any pattern in $x(t)$ useful for decoding and make the signal on the air, $y(t)$, look like white noise. Thus, the random values in $r(t)$ should vary as fast as $x(t)$ –i.e., the bandwidth of $r(t)$ needs to be as large as the bandwidth of the card's data $x(t)$.

To better understand the above point, consider the MBTA Charlie subway card as an example. Fig. 1(a) shows a few bits of the card's reply while communicating with a conventional reader, as perceived by an eavesdropper. The card uses Manchester encoding, where a '0' bit is expressed as a constant value followed by switching repeatedly between two states, whereas a '1' bit is expressed as switching state followed by a constant value. The reader's random signal $r(t)$ when multiplied by $x(t)$ should destroy these internal patterns of the card's reflection. Hence, $r(t)$ has to change faster than any transition in the card's signal. Since the card's data has a bandwidth slightly less than 2 MHz, $r(t)$ should span a bandwidth of 2 MHz.

In our design, the RF-Cloak reader generates a sequence of 2 million random complex samples per second drawn from a complex Gaussian distribution with a variance equal to the average transmission power of the reader. Given this random modulation, Fig. 1(b) shows the time signal received by the eavesdropper for the same bits as in Fig. 1(a). Both the '0' bits and the '1' bits are now dispersed by the rapidly changing $r(t)$ and hence have the appearance of random white noise on the air. The eavesdropper can no longer distinguish them to decode. Additionally, Fig. 1(c) shows the frequency profile of the eavesdropper's received signal, which exhibits a flat profile characterizing white noise spanning 2 MHz.

We analytically show that even if the eavesdropper uses the optimal decoder (i.e., the maximum likelihood



(a) Charlie card communication

(b) Random modulation of Charlie card's signal

(c) Frequency profile of randomly modulated signal

Figure 1—The signal at the eavesdropper during the MBTA subway card's reply: (a) shows the eavesdropper's received time signal when the card communicates '1001' to a conventional RFID reader. Two patterns are used to disambiguate '0' and '1'. (b) shows the received signal when the random modulation $r(t)$ varies faster than the rate of the card. (c) plots the frequency profile of the randomly modulated card's signal, which is flat like white noise.

decoder), his bit error rate will be close to 50% which is no better than randomly trying to guess the bits of the RFID's data. Specifically, in Appendix A, we derive the eavesdropper's optimal decoder and prove the following lemma about RF-Cloak's random modulation.

LEMMA 4.1. *There exists a constant $C < 1$ such that given a random signal $r(t)$ whose samples are drawn from a complex Gaussian distribution with zero mean, and whose bandwidth is as large as $x(t)$, a single antenna eavesdropper using the optimal decoder achieves a bit error rate (BER) in decoding $x(t)$ of:*

$$BER = \frac{1}{2} - \epsilon \quad where \quad \epsilon < C \cdot \sqrt{\frac{Power\ of\ RFID\text{'s signal}}{Power\ of\ Reader\text{'s signal}}}$$

Since the power reflected by the RFID is much weaker than the reader's direct signal power, $\epsilon \approx 0$ and the BER $\approx 1/2$. For typical scenarios, the card's reflected signal is 20 to 30 dB weaker than the reader's RF signal [7, 18, 53]. Hence, the eavesdropper's BER assuming no channel noise is around 40%–47%. Further, our empirical results in §6.1 show that the eavesdropper's mean BER is 49.8%. This higher BER is because in practice the wireless channel noise exacerbates the BER.

4.2 How Does the RF-Cloak Reader Decode?

The goal of the RF-Cloak reader's decoder is to retrieve the card's data $x(t)$ from the received signal $y(t)$. The reader received signal is:

$$y(t) = h_{reader \to self} \cdot r(t) + h_{card \to reader} \cdot x(t) \cdot r(t), \quad (3)$$

where $h_{reader \to self}$ is the channel of the reader's self interference, and $h_{card \to reader}$ is the channel of the card's reflection at the reader.

To decode, the RF-Cloak reader needs to eliminate the effect of the random signal $r(t)$ in Eq. 3 to obtain $x(t)$. The first term in the above equation, $h_{reader \to self} \cdot r(t)$, is the reader's self-interference over the wire. Canceling self-interference is a standard procedure in RFID readers [15, 57] since the reader has to receive the tag's signal while transmitting its own signal (without which the RFID tag cannot transmit). The reader cancels its self-interference using a device called circulator [33], which eliminates most of the signal in the analog domain. It then processes the signal in the digital domain to eliminate any residual self-interference. This is done by subtracting $h_{reader \to self} \cdot r(t)$ from the received signal $y(t)$. The reader knows $r(t)$ since it generated the random signal. As for the channel, $h_{reader \to self}$, it is estimated using standard channel estimation methods [30].

Removing the self-interference term from Eq. 3 yields:

$$\widehat{y}(t) = \quad h_{card \to reader} \cdot x(t) \cdot r(t) \quad (4)$$

Next, the reader divides $\widehat{y}(t)$ by $h_{card \to reader} \cdot r(t)$, which produces $x(t)$.[3] The reader can do so because it knows $r(t)$ and can compute the channel $h_{card \to reader}$ using the known preamble in the card message. Once the reader has $x(t)$, it decodes the data bits as in standard RFID decoding.

5. RF-Cloak: Randomized Channel

In this section, we focus on defending against MIMO (multi-input multi-output) eavesdroppers. The challenge in securing RFIDs against MIMO adversaries stems from the fact that a MIMO system with n receivers can separate (and independently decode) n signals transmitted concurrently on the wireless medium [23, 36]. Thus, a 2-receiver MIMO eavesdropper can separate the reader's random modulation from the card's signal, and decode the latter. Below we explain this challenge in detail and design a solution that overcomes MIMO eavesdroppers.

5.1 Challenge: The MIMO Game

MIMO transforms the RFID eavesdropping problem into a game between the eavesdropper and the reader:

[3] Dividing a noisy received signal by $r(t)$ can potentially increase the noise variance, due to the random structure of $r(t)$. One way to refine the decoding at low SNRs is to use a matched filter and correlate with $r(t)$ [24].

if the eavesdropper has a larger MIMO system than the reader, it can separate the reader's random signal from the RFID's signal and decode the latter. Thus, with random modulation alone, to win this game, the reader needs to keep adding MIMO transmitters to match or exceed the number of receivers on the MIMO eavesdropper. For example, in §4, we demonstrated that a single-transmitter reader transmitting a random signal, $r(t)$, can defend against a single-receiver eavesdropper. Let us examine, what happens if the reader continues to use one transmitter but the eavesdropper upgrades to a 2-receiver MIMO system.

A 2-receiver MIMO eavesdropper receives two signals, $y_1(t)$ and $y_2(t)$:

$$\begin{aligned} y_1(t) &= (h_{reader \rightarrow eve1} + h_{card \rightarrow eve1} \cdot x(t)) \cdot r(t) \\ y_2(t) &= (h_{reader \rightarrow eve2} + h_{card \rightarrow eve2} \cdot x(t)) \cdot r(t), \end{aligned} \tag{5}$$

where $h_{reader \rightarrow eve1}$ and $h_{reader \rightarrow eve2}$ are the channels from the reader to the eavesdropper's first and second receivers respectively, and $h_{card \rightarrow eve1}$ and $h_{card \rightarrow eve2}$ are the channels of the card's reflected signal at the eavesdropper's receivers.

The MIMO eavesdropper can first eliminate the random multiplier $r(t)$ by dividing the two signals he receives:

$$\frac{y_1(t)}{y_2(t)} = \frac{h_{reader \rightarrow eve1} + h_{card \rightarrow eve1} \cdot x(t)}{h_{reader \rightarrow eve2} + h_{card \rightarrow eve2} \cdot x(t)}. \tag{6}$$

Next, the eavesdropper tries to decode $x(t)$ from Eq. 6, which has no random multiplier. Recall that the card's message $x(t)$ has only two states: $x(t) = x_0$ when the card's load is *off* (i.e., open circuit), and $x(t) = x_1$ when the card's load is *on* (i.e., reflecting the reader's signal). Distinguishing these two states enables the eavesdropper to track the state transition and decode the card's transmitted data $x(t)$. Note that the ratio of the received signals in Eq. 6 takes only two values corresponding to the $x(t) = x_0$ state and the $x(t) = x_1$ state. We denote these two values of the ratio y_1/y_2 as α_0 and α_1. Thus, after computing the ratio y_1/y_2, the only ambiguity the eavesdropper has is in mapping the two observed values α_0 and α_1 to states x_0 and x_1. To resolve this ambiguity, the attacker checks which of the two mappings allows the decoded message to satisfy the checksum [19]. Thus, a 2-receiver MIMO eavesdropper can win the MIMO game over a single-transmitter reader, even if the latter uses random modulation.[4]

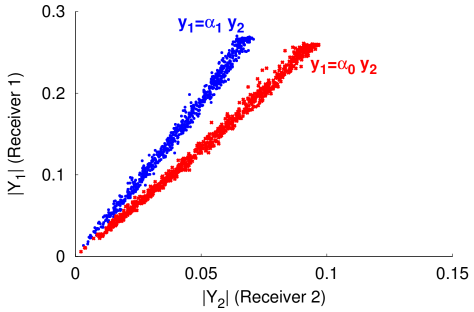


Figure 2—2-Dimensional space of a 2-receiver MIMO eavesdropper in RF-Cloak's random modulation scheme: The figure shows a scatter plot of the samples received by a 2-receiver eavesdropper. Despite random modulation, a 2-receiver eavesdropper facing a single-transmitter reader sees two lines corresponding to two states of the RFID card, x_0 and x_1 which allows it to decode.

We can gain a deeper insight into this MIMO game by looking at the received signal in a 2-dimensional space created by the two receivers on the eavesdropper, where one dimension is $y_1(t)$, the signal received on his first receiver and the other dimension is $y_2(t)$, the signal received on his second receiver. At any point in time t, the received signals $(y_1(t), y_2(t))$ can be represented as one point in this 2-dimensional space. When $x(t) = x_0$, we know from above that $y_1 = \alpha_0 y_2$, which defines a *line* in this 2-dimensional space. Similarly, when $x(t) = x_1$, the received signals lie on a different line defined by $y_1 = \alpha_1 y_2$.

We confirm this point empirically by letting a 2-receiver MIMO adversary (implemented using USRP software radios) eavesdrop on a conversation between a commercial UHF RFID and a USRP-based reader that employs random modulation. Fig. 2 shows a scatter plot of what the eavesdropper receives on its two antennas. Here, we plot the magnitude of the received samples, i.e., each point in the figure represents $(|y_1(t)|,|y_2(t)|)$ for a specific t. We then use our ground truth knowledge of the actual bits transmitted by the RFID card to label samples corresponding to x_0 in blue and x_1 in red. Despite the fact that the received signal at each receiver is random, together $y_1(t)$ and $y_2(t)$ span only *lines* instead of the entire 2-dimensional space at the eavesdropper. Since the card's data has only two states, we see two lines in the figure and hence the eavesdropper can decode by checking which line the received samples belong to.

The above can be generalized to larger-scale MIMO systems on the reader and eavesdropper. If the eavesdropper has n receive chains, he receives signals in an n-dimensional space. If the reader has k transmit chains ($k < n$) and transmits k signals from them, these signals will only span a k-dimensional *subspace* (lines, planes, etc.) in the eavesdropper's n-dimensional space. Since the card has only two states x_0, x_1, the eavesdropper will observe two unique subspaces and hence he can decode. Thus, it comes down to a MIMO game between the reader and the eavesdropper. No matter how many transmit chains the reader uses, the eavesdropper can win the game by using more receive chains.

[4] Note that the eavesdropper is able to decode without having to estimate any of the wireless channels in Eq. 5.

5.2 Change the Game: Channel Randomization

To overcome the MIMO game, let us go back to Fig. 2 and try to understand why we have separate slopes for the two states of the RFID signal. Recall that the slopes of the two lines in Fig. 2, α_0 and α_1, depend only on the channels from the reader to the eavesdropper receivers, as clear from Eq. 6. If the channels stay constant, the two lines $y_2 = \alpha_0 y_1$ and $y_2 = \alpha_1 y_1$ in Fig. 2 do not change over time. However, if the channels from the reader to the eavesdropper's MIMO antennas are random, then the ratio y_1/y_2 will be random and the slope will be random for every transition in the state of the RFID's signal. This prevents the eavesdropper from separating the points corresponding to the x_0 state of the RFID from the points corresponding to the x_1 state of the RFID. Thus, we can overcome a MIMO eavesdropper by randomizing the wireless channels to the eavesdropper.

We analytically show that if the channels from the reader to the eavesdropper are random, a MIMO eavesdropper that uses the optimal decoder (maximum likelihood decoder) will see a bit error rate close to 50%, which is no better than a random guess. Specifically, in Appendix B we prove the following lemma:

LEMMA 5.1. *There exists a constant $C < 1$ such that, given the wireless channels from the reader to the eavesdropper's antennas are random complex Gaussians with zero mean and the channels change as fast as the bandwidth of $x(t)$, a MIMO eavesdropper with n receivers using the optimal decoder achieves a bit error rate (BER):*

$$BER = \frac{1}{2} - \epsilon \ \ where \ \ \epsilon < C\sqrt{n} \cdot \frac{Power\ of\ RFID's\ signal}{Power\ of\ Reader's\ signal}$$

Recall that the power reflected by the RFID is much weaker than the reader's direct signal power. For a typical power ratio of −30 dB to −20 dB [56], assuming no channel noise, even a 20 antenna MIMO eavesdropper will have BER around 48% to 49.8%.

To build a system that randomizes the channels, RF-Cloak uses a combination of antenna motion and random rapid antenna switching. Specifically, past work shows that due to multipath effects, even small motion of the antenna can create large variations in the wireless channels [2, 41, 50, 54]. Hence, by leveraging antenna motion, we are able to span a large range of random channel instantiations. We further increase the randomization by combining antenna motion with rapid and random switching of antennas. Specifically, we use a rotating frame that holds multiple antennas, and we randomly switch between the antennas using rapid switches [20] that can switch every few microseconds. Random switching breaks the periodicity of rotation as well as any correlation in the channel instantiations over time. Note that while our reader uses switched antennas, it is not a MIMO system because it has only *one transmit/receive chain*, to which all antennas are connected via a switch.

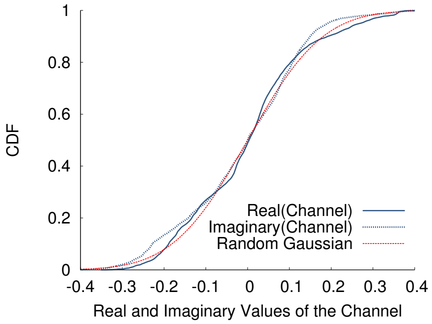


Figure 3—Distribution of the channel seen at MIMO eavesdropper receiver: This figure shows the CDF of the distribution of the real and imaginary part of RF-Cloak reader's random channel to one receiver of the MIMO eavesdropper. The real and imaginary parts match a random Gaussian distribution with zero mean and standard deviation $\sigma = 0.1414$ This shows that the channel is random and spans a large range of values.

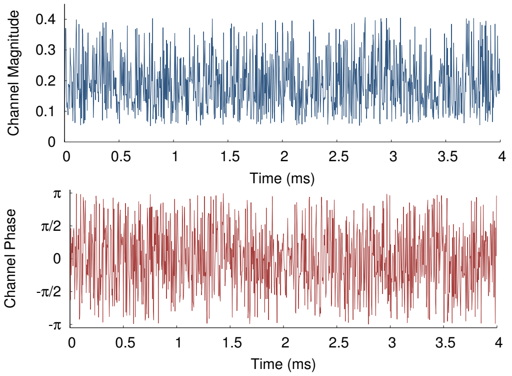


Figure 4—Channel randomization at MIMO eavesdropper receiver: This figure shows RF-Cloak reader's random channel to one receiver of the MIMO eavesdropper. Due to the rapid and random switching of the antennas together with the antenna motion, each eavesdropper receiver sees a large number of randomly and rapidly changing channels (both magnitude and phase), which undermines the eavesdropper's MIMO decoding capability.

Fig. 3 shows the channel resulting from this system at one of the eavesdropper's MIMO receivers. The figure plots the distributions of the real and imaginary parts of the channel instantiations observed over a period of 4 ms. The figure shows that the distributions matches a random Gaussian distribution with zero mean. This demonstrates that our implementation of channel randomization has produced random Gaussian channel instantiations, even when the channel is observed over a short interval of 4 ms. Fig. 4 shows the magnitude and phase of the channel as functions of time over the same 4 ms, showing that they are randomly switched at high speed.

To gain a deeper insight into how randomizing the channel prevents the eavesdropper from decoding, we again go back to Fig. 2. We repeat the same exper-

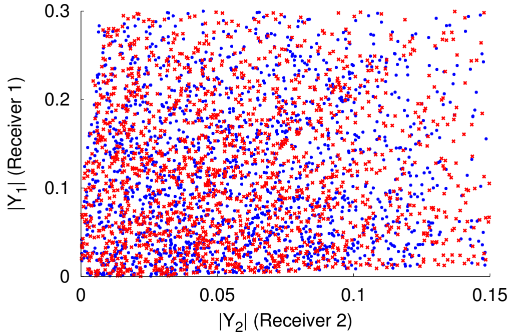


Figure 5—2-Dimensional space of the 2-receiver MIMO eavesdropper when the reader randomizes the channels: The eavesdropper's received samples $(|Y_1|, |Y_2|)$ almost span the entire space. No subspace is unique to the card's x_0 state (red) as opposed to the x_1 state (blue), which prevents the eavesdropper from decoding.

iment with the 2-receiver MIMO eavesdropper. However, this time we replace the reader's static antenna with the aforementioned channel randomization setup. Fig. 5 shows the scatter plot of the two signals received by the 2-receiver MIMO eavesdropper. In contrast to Fig. 2, now the received signal samples span the entire space, instead of being confined to two lines. Hence, the eavesdropper in this case cannot tell apart the blue points and the red points and cannot decode the RFID's message.

5.3 How Does the RF-Cloak Reader Decode?

The RF-Cloak reader needs to be able to retrieve the card's data despite the channel randomization. The reader receives the signal:

$$y(t) = h_{reader\to self} \cdot r(t) + h_{card\to reader}(t) \cdot x(t) \cdot r(t), \quad (7)$$

where $h_{reader\to self}$ is the reader's self-interference channel and $h_{card\to reader}(t)$ is the channel of the card's reflected signal at the reader. The reader can cancel its self interference $h_{reader\to self} \cdot r(t)$ as described in §4.2. Note that $h_{reader\to self}$ is not random since it is the channel from the antenna to itself over the wire and hence it is not affected by motion. Once the reader eliminates its self interference and the random modulation $r(t)$, what remains is:

$$\widehat{y}(t) = h_{card\to reader}(t) \cdot x(t) \quad (8)$$

Since $h_{card\to reader}(t)$ is random and cannot be estimated, the reader needs to decode based on the power. Recall that, when the card switches off its reflection via an open circuit, its state $x_0 \approx 0$. And hence, by detecting the power when the card's signal is in the x_1 state, the reader can distinguish the two states and decode. In Appendix C, we derive the optimal decoder and BER and in §6.2 we empirically show that RF-Cloak can decode the RFID's data.

6. Implementation & Evaluation

We built a prototype of RF-Cloak using USRP software radios [32] and used it to secure the communication of off-the-shelf RFID cards. We adopt a UHF reader code base developed in [11] and extend it to also work with HF RFIDs.

To randomize the modulation, we customize the reader software to transmit a random signal generated as described in §4 instead of a constant waveform, during the card's reply. For channel randomization, we connect the reader's single transmit chain to 8 antennas using a programmable switch and randomly switch between them at the same rate as the card switches between its *on* and *off* states. The switch is built using three off-the-shelf multiplexers [20] controlled by a programmable micro-controller [6]. Furthermore, the transmit antennas are mounted on a circular frame which is rotated by a 1725 RPM fan motor.

A. UHF Devices

Reader: The UHF RF-Cloak reader is built using USRP N210 with RFX900 daughterboards and VERT900 omnidirectional antennas.

RFID Card: We use the Alien Squiggle General Purpose RFID Tags [4] as an example of UHF passive RFIDs.

Eavesdropper: The eavesdropper is implemented using the same hardware (USRP and antenna) as the RF-Cloak reader. The only difference is that, in the MIMO experiments, the eavesdropper uses multiple (up to 5) USRPs and receive antennas distributed across space.

B. HF Devices

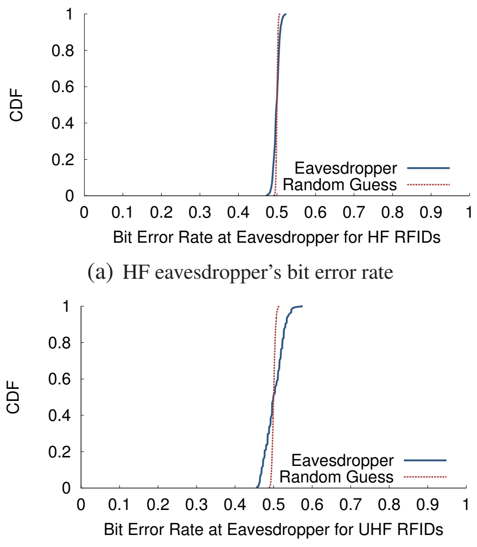
Reader: The HF RF-Cloak reader is implemented using USRP1 software radio with LFTX and LFRX daughterboards operating in the 0-30 MHz frequency range and the DLP-RFID-ANT antennas [17].

RFID Card: We use the MBTA Charlie card [46] as an example of the widely used MIFARE Classic cards.

Eavesdropper: The eavesdropper is implemented using the same hardware (USRP and antenna) as the RF-Cloak reader.

C. Security Metric

We use the bit error rate (BER) experienced by the eavesdropper as a metric for the system's security. Ideally, a fully secure system should maintain a 50% BER at the eavesdropper, which is equivalent to the result of a random guess. For both HF and UHF RFIDs, we run experiments at a variety of reader, card, and eavesdropper locations and average across 1000 runs to compute the mean BER for each placement .



(a) HF eavesdropper's bit error rate

(b) UHF eavesdropper's bit error rate

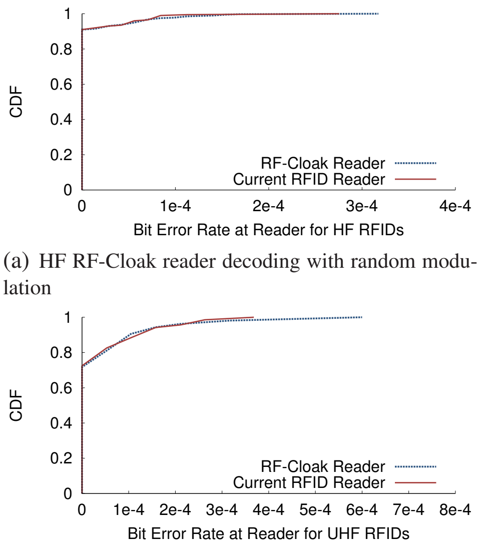
Figure 6—Effectiveness of random modulation against single-antenna eavesdroppers: CDF of the eavesdropper's BER. (a) For HF cards, the eavesdropper's BER closely matches a random guess. (b) For UHF cards, the eavesdropper's average BER is 50.3% with a standard deviation of 2.3%.

6.1 Evaluation of Randomized Modulation

First, we investigate whether RF-Cloak's random modulation can protect HF and UHF RFIDs from a single-receiver eavesdropper.

Experiment: The RF-Cloak reader queries the Charlie card or the commercial UHF tag for 1000 times in each run. To match the operating range in current RFID systems, the distance between the RF-Cloak reader and the RFID card is varied between 2–10 cm in the HF case, and 1–5 meters in the UHF case. During the RFID's reply, the reader continuously transmits a random signal generated using the design in §4. In the case of the Charlie card (HF), the eavesdropper is placed 5–10 cm away from the card; in the UHF case, it is placed 0.2–5 meters away from the UHF RFID tag. The eavesdropper has a single receive chain and a single antenna. It tries to decode the tag's message using the maximum-likelihood decoder described in Appendix A.

Result 1 (BER at the Eavesdropper): Fig. 6(a) plots the CDF of the eavesdropper's bit error rates when the Charlie card is communicating with an RF-Cloak reader. The CDF is taken over all locations of the reader, Charlie card, and eavesdropper. For comparison, the red dashed curve is the CDF of the eavesdropper's BER when it randomly guesses the bits without trying to make use of the eavesdropped information. The figure shows that, when the RF-Cloak reader randomizes the modulation, the eavesdropper's BER is 49.8% on average, with a standard deviation of less than 0.8%, closely matching the result of a random guess.



(a) HF RF-Cloak reader decoding with random modulation

(b) UHF RF-Cloak reader decoding with random modulation

Figure 7—RF-Cloak reader's decoding with random modulation: (a) For the HF cards, the average BER of the reader is less than 0.01% with a maximum of 0.03%. (b) For UHF cards, the average BER of the reader is less than 0.01% with a maximum of 0.06%. Hence, the decoding performance of the RF-Cloak reader is on par with that of existing readers.

Similarly, Fig. 6(b) plots the CDF of the UHF eavesdropper's BER. Due to the significantly larger range in UHF systems, the BER has a slightly higher standard deviation than HF systems. The UHF eavesdropper's BER is 50.3% on average with a standard deviation of 2.3%. Thus, RF-Cloak's random modulation renders the decoding at the eavesdropper no better than a random guess.

Result 2 (Decoding Performance of RF-Cloak Reader): Next, we verify that replacing the constant waveform with RF-Cloak's randomized modulation does not affect the decoding at the reader. We use the signals from the same experiment above but now focus on the reader's decoding BER.

Fig. 7(a) and Fig. 7(b) show the CDFs of the bit error rates at the RF-Cloak reader for the HF and UHF experiments respectively. For reference, the figure also shows the bit error rates of existing RFID readers that use a constant waveform instead of the random modulation, for the same placements of reader and card. The HF RF-Cloak reader has an average decoding BER of less than 0.01% and a maximum BER of 0.03%, whereas the UHF RF-Cloak reader has an average bit error rate of less than 0.01% and a maximum of 0.06%. These bit error rates are typical for RFID systems and on par with current RFID reader's performance.

6.2 Evaluating RF-Cloak with MIMO Eavesdroppers

Next, we study RF-Cloak's capability of protecting

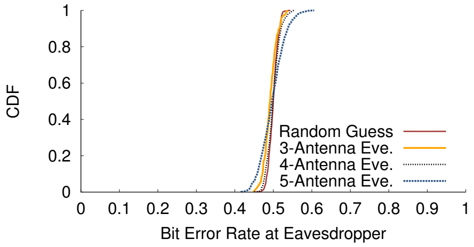


Figure 8—Effectiveness of channel randomization in defending against MIMO eavesdroppers: CDF of the MIMO eavesdropper's BER when the RF-Cloak reader randomizes its channels to the eavesdropper via antenna switching and motion. The BER is on average 50% and is very close to a random guess even if the eavesdropper uses 3, 4, or 5 receivers.

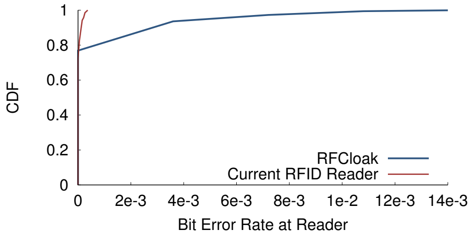


Figure 9—Decoding performance of RF-Cloak reader with channel randomization: The average BER at the RF-Cloak reader with antenna switching and rotation is 0.2%, which is fairly close to the performance of current RFID readers.

RFIDs from a MIMO eavesdropper employing multiple receive chains and antennas. Note that MIMO does not benefit eavesdroppers in HF RFID systems for the following reason. The ability of a MIMO eavesdropper to separate the reader's random signal from the RFID's signal hinges on the channels he perceives from the reader and the RFID being sufficiently different. However, in HF (13.56 MHz) RFID systems, the operating distance between the card and the reader is within 10 cm, significantly smaller than the wavelength (22 meters). In this case, it is well-known that MIMO techniques cannot separate their signals [65]. Hence, here we focus on UHF RFIDs in our evaluation with MIMO eavesdroppers.

Experiment (MIMO & Channel Randomization): We repeat the same experiment performed in the previous section, after replacing the single-antenna eavesdropper by a MIMO eavesdropper and introducing channel randomization at the RF-Cloak reader, using one transmit chain with random antenna switching and rotation as described in §6. We vary the number of receive chains and antennas employed by the MIMO eavesdropper between 3, 4, and 5. The eavesdropper decodes as described in Appendix B.

Result 1 (MIMO Eavesdropper v.s. Channel Randomization): Fig. 8 plots CDFs of the BER experienced by 3- 4- and 5-antenna MIMO eavesdroppers when the RF-Cloak reader uses channel randomization. For reference, the BER result of a random guess is also plotted. The figure shows that the eavesdropper experiences a BER close to 50%, with a standard deviation ranging between 1.2% and 2.9%, depending on the number of receivers at the eavesdropper. Hence, the eavesdropper's decoding in face of RF-Cloak's channel randomization scheme is equivalent to a random guess. This is because the samples corresponding to x_0 and x_1 states are now indistinguishable in the multi-dimensional space.

Result 2 (Decoding Performance of the RF-Cloak Reader with Channel Randomization): Finally, we verify that the antenna switching/motion and the resulting channel randomization do not prevent the trusted RF-Cloak reader from decoding. Fig. 9 shows the BER from the same experiment as above but as perceived by an RF-Cloak reader that decodes the signal using our design in §5.2. As we can see, the RF-Cloak reader has an average decoding bit error rate of 0.2%. Note that the RFID packet length is typically short, since most of the communication involves transmitting 16-bit temporary IDs plus 5-bit checksum. In this case, a 0.2% bit error rate translates into a packet loss rate of around 4%, which is quite common and acceptable in RFID systems. If certain applications require an even lower BER, the reader can request the tags to transmit their data using longer codes, an option readily available in today's commercial RFIDs [19].

In conclusion, RF-Cloak's channel randomization via rapid antenna switching and motion provides an effective mechanism to protect RFIDs from MIMO eavesdropping, without requiring MIMO capability at the reader.

7. Related Work

Past work on defending RFIDs against eavesdropping has mainly focused on improving the cryptographic protocols [1, 9, 13]. These schemes, however, are difficult to build in practice due to the severe energy, size and cost constraints on RFID cards. Thus, commercial RFIDs continue to use weak encryption schemes proven to be vulnerable [38, 51, 63].

RF-Cloak belongs to the class of physical layer security mechanisms that aim to defend against eavesdroppers without modifying the RFIDs. The closest to our work is the Noisy Reader proposal [60], in which the reader varies its own signal in an attempt to hide an HF RFID's data. It generates one random number per card bit and uses it as the magnitude of the reader's signal. It also tries to imitate the card's internal bit pattern by making the reader periodically switch its signal phase by 180° at the same frequency the card switches between two states. The Noisy Reader scheme was studied analytically, yet we are unaware of any prior implementation or empirical evaluation. We implemented the Noisy Reader using the same USRP setup as RF-Cloak. Fig. 10

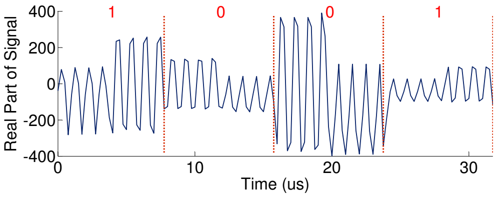


Figure 10—Noisy Reader trace: The eavesdropper's received signal of the Charlie card communicating with the Noisy Reader still exhibits two clear patterns corresponding to the '0' bits and '1' bits. Despite the random magnitude in each bit and the phase shifting, the eavesdropper can still decode by comparing the first half and the second half of each bit. The '0' bits have the same shape, while the '1' bits have a different one.

shows the received signal at a single-antenna eavesdropper, when the Noisy Reader is protecting the Charlie card. Although each bit is scaled differently, we can still see that all the '0' bits have the same shape, while the '1' bits have a different shape. This is due to the multiplicative nature of the card's signal and the Manchester encoding shown in Fig. 1(a) which is used by more than 80% of the HF cards today (ISO 14443 Type A [51]). Our experiments show that a single-antenna eavesdropper is able to fully decode the Charlie card's data in 99.7% of the traces despite the Noisy Reader.

Another prior work in this category, BUPLE [14], tries to hide the RFID's message using frequency hopping at the reader. However, given the frequency band that commercial backscatter RFIDs operate in (i.e., 902 MHz – 928 MHz), any typical receiver (e.g., USRP) with a bandwidth larger than 26 MHz can easily identify the center frequency at any point in time and decode the RFID's signal. Other physical layer solutions to eavesdropping attacks, such as the Noisy Tag [12], require modifying the cards to use wireless signals to exchange a key with the reader.

Past theoretical work from the information theory community has also explored the use of antenna switching for secure physical layer communication [3, 16, 31, 42, 66]. These papers use large switched antenna arrays to maintain a decodable signal towards the direction of the intended receiver (i.e., a constant main beam of the antenna array), but scramble the signal at undesired directions (i.e., sidelobes of the array pattern) to prevent the eavesdropper from decoding. Such techniques do not work in the context of passive RFID communication, where the RFID reflects the reader's signal to all directions regardless of the reader signal's directionality.

RF-Cloak also builds on jamming-based systems [25, 60, 62]. However, these solutions use standard jamming and cannot be applied directly to RFIDs. Standard jamming deals with wireless devices that transmit their own signal, in which case the random jamming signal *adds up* to the protected data. RFIDs, on the other hand, reflect the reader's signal without transmitting a signal of their own. Hence, the random signal *multiples with* the protected data. Because of this multiplicative model, directly applying jamming to RFIDs yields insecure systems like in the case of the Noisy Reader [60] described above in details.

Our work is also related to Near Field Communication (NFC) security on mobile phones [27, 48, 70]. These systems, however, operate in very close proximity and are not applicable to UHF RFIDs that operate at a distance of few meters away from the reader. RF-Cloak provides a solution that is applicable to both UHF RFIDs as well as near field HF RFIDs.

Finally, antenna motion has been recently exploited in wireless communication for interference management [2] as well as RF localization [39, 40, 47, 61, 68, 69] and WiFi Imaging [26]. Differing from these, RF-Cloak leverages antenna motion to randomize the wireless channels and enable a security construct for defending against MIMO eavesdropping.

8. Conclusion

Recent eavesdropping attacks have compromised the security of billions of deployed RFIDs worldwide. This paper asks whether one can secure these simple RFIDs from eavesdropping attacks, without modifying the cards. By only implementing changes on the RFID reader, RF-Cloak introduces random modulation and random channels to overcome powerful MIMO eavesdroppers. We demonstrated that randomizing the modulation via reflection, and randomizing the wireless channels by using antenna motion and rapid switching can effectively protect today's widely used commercial RFIDs from eavesdroppers. Further, we believe the channel randomization technique can be combined with many existing security primitives, which opens doors to a variety of new designs in wireless security beyond the scope of RFID communication.

Acknowledgments:

The authors would like to thank the shepherd Deepak Ganesan as well as Lixin Shi, Fadel Adib, Deepak Vassisht, Badih Ghazi, Swarun Kumar, and Hariharan Rahul for their insightful feedback and comments.

Appendix

A. Proof of Lemma 4.1

The eavesdropper receives the signal $y(t)$ in Eq. 2. Since $h_{reader\to eve}$ is constant, we can normalize $y(t)$ by it to get:[5]

$$y'(t) = r(t) \cdot \left[1 + \frac{h_{card\to eve}}{h_{reader\to eve}} \cdot x(t)\right] \tag{9}$$

[5] In this derivation, we ignore wireless channel noise, since it will only increases the BER of the eavesdropper.

The RFID card's signal $x(t)$ has two states: x_0 when the card has an open circuit and x_1 when the card turns on its load to reflect the reader's signal. To convey a '0' or '1' bit, the card transmits different patterns of x_0's and x_1's of length k. Thus, for each card bit b, the eavesdropper receives k samples in $y'(t)$ denoted as $\{Y_1, Y_2, \cdots, Y_k\}$:

$$Y_i = \begin{cases} R_i \cdot (1 + p_i^0) & \text{if } b = 0 \\ R_i \cdot (1 + p_i^1) & \text{if } b = 1 \end{cases} \tag{10}$$

where $\{p_1^0, ..., p_k^0\}$ is the pattern when the card transmits a '0' bit and $\{p_1^1, ..., p_k^1\}$ is the pattern when the card transmits a '1' bit.[6] R_i is a sample in the reader's random signal $r(t)$ which is drawn from a complex normal distribution with zero mean and standard deviation σ. Note that, since the bandwidth of $r(t)$ is the same as $x(t)$, there is a single R_i for each state of the RFID's signal. We will assume the eavesdropper knows the bit boundaries i.e. he knows which Y_i samples correspond to the same bit.

The eavesdropper's optimal decoder is a maximum likelihood decoder as derived in [8, 35]. The optimal decoder is the one that achieves the minimum bit error rate. Hence, an eavesdropper using any other strategy cannot extract more information than an eavesdropper using the optimal decoder. Given the k received samples $\{Y_1, Y_2, \cdots, Y_k\}$ at the eavesdropper, the decoder is defined by the following hypothesis test:

$$Pr(b = 1|\{Y_1, \cdots, Y_k\}) \quad \overset{1}{\underset{0}{\substack{\geq \\ <}}} \quad Pr(b = 0|\{Y_1, \cdots, Y_k\})$$

Because the card's bits have equal probability of being '0' or '1' [19, 51], we can rewrite the hypothesis test as:

$$Pr(\{Y_1, \cdots, Y_k\}|b = 1) \quad \overset{1}{\underset{0}{\substack{\geq \\ <}}} \quad Pr(\{Y_1, \cdots, Y_k\}|b = 0)$$

Given $b = 0$ or $b = 1$, the k samples in $\{Y_1, \cdots, Y_k\}$ become independent Gaussians with zero mean and standard deviation $\sigma_i^0 = \sigma|1+p_i^0|$ or $\sigma_i^1 = \sigma|1+p_i^1|$. Hence, we can write:

$$Pr(Y|b = 0) = \frac{1}{(2\pi)^{k/2} \prod \sigma_i^0} \cdot \exp\left(-\sum_i^k \left(\frac{|Y_i|}{\sigma_i^0} \right)^2 \right)$$

A similar equation can be derived for $b = 1$. Since the two patterns have the same number of x_0 samples, we have $\prod \sigma_i^0 = \prod \sigma_i^1$. The maximum-likelihood decoder can then be simplified to:

$$\sum_i^k \left(\frac{|Y_i|}{\sigma_i^1} \right)^2 \quad \overset{1}{\underset{0}{\substack{\leq \\ >}}} \quad \sum_i^k \left(\frac{|Y_i|}{\sigma_i^0} \right)^2$$

Given the patterns p^0 and p^1 for UHF RFIDs [60], we can further simplify the UHF decoder to:

$$|Y_{10}|^2+|Y_{12}|^2+|Y_{14}|^2+|Y_{16}|^2 \quad \overset{1}{\underset{0}{\substack{\leq \\ >}}} \quad |Y_9|^2+|Y_{11}|^2+|Y_{13}|^2+|Y_{15}|^2$$

Similarly, given the patterns for HF RFIDs [60], we can simplify the HF decoder to:

$$|Y_2|^2+|Y_4|^2+|Y_6|^2+|Y_8|^2 \quad \overset{1}{\underset{0}{\substack{\leq \\ >}}} \quad |Y_9|^2+|Y_{11}|^2+|Y_{13}|^2+|Y_{15}|^2$$

Given the above optimal decoders, we derive the bit error rate (BER) at the eavesdropper for the case of UHF RFID cards. The derivation is the same for the HF RFID cards.

Define the random variables U, V, and Z such that $U = |Y_{10}|^2 + |Y_{12}|^2 + |Y_{14}|^2 + |Y_{16}|^2$, $V = |Y_9|^2 + |Y_{11}|^2 + |Y_{13}|^2 + |Y_{15}|^2$, and $Z = U - V$. Then, the bit error rate at the eavesdropper is defined as:

$$BER = \frac{1}{2} Pr(Z < 0|b = 0) + \frac{1}{2} Pr(Z > 0|b = 1) \tag{11}$$

Given $b = 0$, $\{Y_2, Y_4, Y_6, Y_8\}$ are independent complex gaussain random variables with zero mean and standard deviation $\sigma_U = \sigma(1 + x_1)$ while $\{Y_9, Y_{11}, Y_{13}, Y_{15}\}$ are the same but with standard deviation $\sigma_V = \sigma(1 + x_0)$. Thus, U and V have a Gamma distribution with degree 4 and rate of $2\sigma_u^2$ and $2\sigma_v^2$ [5]. We can now derive the distribution of Z for $z \leq 0$ as:

$$\begin{aligned} \underset{Z}{\Pr}(z|b = 0) &= \int_0^\infty \underset{U}{\Pr}(u) \cdot \underset{V}{\Pr}(u - z) du \\ &= \frac{256 \cdot \sigma_U^8 \sigma_V^8}{\beta^4} \left(\frac{20}{\beta^3} - \frac{10z}{\beta^2} + \frac{2z^2}{\beta} - \frac{z^3}{6} \right) e^{2\sigma_V^2 z} \end{aligned}$$

where $\beta = 2\sigma_U^2 + 2\sigma_V^2$. In a similar manner, we can derive the distribution of Z given $b = 1$ for $z \geq 0$. We can now integrate to calculate the probabilities in Eq. 11 and the BER as

$$BER = 1 - \frac{\mu^4}{(1+\mu)^4} \left(\frac{20}{(1+\mu)^3} + \frac{10}{(1+\mu)^2} + \frac{4}{1+\mu} + 1 \right)$$

where $\mu = (1+x_1)^2/(1+x_0)^2$. Since $x_0 \approx 0$ and $x_1 \ll 1$, we get that $1/(1 + \mu) \approx 1/(2(1 + x_1))$. Using this, we can rewrite the above BER equation as:

$$BER = \frac{1}{2} - \epsilon \quad \text{where} \quad \epsilon < \frac{29}{32} x_1$$

Recall that, x_1 is the fraction of the reader's signal reflected by the RFID. Hence, $x_1 = \sqrt{\frac{\text{Power of RFID's signal}}{\text{Power of Reader's signal}}}$

B. Proof of Lemma 5.1

An Eavesdropper with n antennas receives n signals $y_1(t), \cdots, y_n(t)$ on each of its n antennas:

$$\begin{bmatrix} y_1(t) \\ \vdots \\ y_n(t) \end{bmatrix} = \left(\begin{bmatrix} h_{r_1}(t) \\ \vdots \\ h_{r_n}(t) \end{bmatrix} + x(t) \cdot \begin{bmatrix} h_{c_1}(t) \\ \vdots \\ h_{c_n}(t) \end{bmatrix} \right) \cdot r(t)$$

[6] $p_i^0 = \frac{h_{card \to eve}}{h_{reader \to eve}} x_0$ or $\frac{h_{card \to eve}}{h_{reader \to eve}} x_1$ depending on the pattern used by the RFID card. For HF cards the patterns are $p^0 =$ [0101010100000000] and $p^1 =$ [0000000010101010]. For UHF cards with miller 8 encoding the patterns are $p^0 =$ [0101010101010101] and $p^1 =$ [0101010110101010].

where $h_{r_i}(t)$ is the random channel from RF-Cloak's antenna to the eavesdropper's i-th antenna, $h_{c_i}(t)$ is the random channel from RFID card to the Eavesdropper's i-th antenna, $r(t)$ is the random modulation signal, and $x(t)$ is the RFID card's reply which takes two states x_0 and x_1. To simplify the analysis, we will ignore the random modulation $r(t)$ in favor of the eavesdropper.

As described earlier, for each bit b, the RFID transmits a pattern $\{p_1^b, \cdots, p_k^b\}$ where $p_j^b = x_0$ or x_1. Thus, the eavesdropper receives k samples per bit on each of its n antennas:

$$\begin{bmatrix} Y_{11} & \cdots & Y_{1k} \\ \vdots & \ddots & \vdots \\ Y_{n1} & \cdots & Y_{nk} \end{bmatrix} = \begin{bmatrix} H_{r_{11}} & \cdots & H_{r_{1k}} \\ \vdots & \ddots & \vdots \\ H_{r_{1n}} & \cdots & H_{r_{nk}} \end{bmatrix} + \begin{bmatrix} H_{c_{11}} & \cdots & H_{c_{1k}} \\ \vdots & \ddots & \vdots \\ H_{c_{1n}} & \cdots & H_{c_{nk}} \end{bmatrix} \begin{bmatrix} p_1^b & \cdots & 0 \\ \vdots & \ddots & \vdots \\ 0 & \cdots & p_k^b \end{bmatrix}$$

The random channels H_{r_i} and H_{c_i} are independent and follow a complex normal distribution with zero mean and standard deviation σ. Similar, to the random modulation, the optimal decoder is a maximum likelihood decoder based on the following hypothesis test:

$$Pr(b = 1|\{Y_{11}, \cdots, Y_{nk}\}) \quad \underset{0}{\overset{1}{\gtreqless}} \quad Pr(b = 0|\{Y_{11}, \cdots, Y_{nk}\})$$

Since the tags bits have equal probability of being '0' or '1', we can rewrite the above hypothesis test as:

$$Pr(\{Y_{11}, \cdots, Y_{nk}\}|b = 1) \quad \underset{0}{\overset{1}{\gtreqless}} \quad Pr(\{Y_{11}, \cdots, Y_{nk}\}|b = 0)$$

Given $b = 0$ or $b = 1$, the Y_{ij} samples become independent complex Gaussains with zero mean and standard deviation $\sigma_{ij}^b = \sigma\sqrt{1 + |p_j^b|^2}$. Their joint probability is:

$$Pr(Y|b) = \frac{1}{(2\pi)^{nk/2}\prod \sigma_{ij}^b} \cdot \exp\left(-\sum_i^n \sum_j^k \left(\frac{|Y_{ij}|}{\sigma_{ij}^b}\right)^2\right)$$

The hypothesis test can now be simplified to:

$$\sum_i^n \sum_j^k \left(\frac{|Y_{ij}|}{\sigma_{ij}^1}\right)^2 \quad \underset{0}{\overset{1}{\lesseqgtr}} \quad \sum_i^n \sum_j^k \left(\frac{|Y_{ij}|}{\sigma_{ij}^0}\right)^2$$

Substituting the patterns for UHF RFID cards, we get

$$\sum_{j \in \{10,12,14,16\}} \sum_i^n |Y_{ij}|^2 \quad \underset{0}{\overset{1}{\lesseqgtr}} \quad \sum_{j \in \{9,11,13,15\}} \sum_i^n |Y_{ij}|^2$$

Given the above optimal decoder, we can derive the BER of the eavesdropper. Define Z as the difference between the left and right hand sides of the the above hypothesis test. Then, Z is the difference between two random variables of a Gamma distribution with degree $4n$ and rates $2\sigma^2(1 + x_1^2)$ and $2\sigma^2(1 + x_0^2)$. Similar to Appendix A, we derive the distribution of Z use it to calculate the BER as:

$$\begin{aligned} BER = & \frac{1}{2}Pr(Z < 0|b = 0) + \frac{1}{2}Pr(Z > 0|b = 1) \\ = & 1 - \frac{\mu^{4n}}{(1+\mu)^{4n}} \sum_{i=0}^{4n-1} \binom{i + 4n - 1}{4n - 1} \frac{1}{(1+\mu)^i} \end{aligned}$$

where $\mu = (1 + x_1^2)/(1 + x_0^2)$. Since $x_0 \approx 0$ and $x_1 \ll 1$, $1/(1 + \mu)^2 \approx 1/(4(1 + x_1^2))$. Using the fact that $\sum_{i=0}^n \binom{n+i}{n} \frac{1}{2^i} = 2^n$ and Stirling's bounds, we can simplify the BER to:

$$BER = \frac{1}{2} - \epsilon \quad \text{where} \quad \epsilon < \frac{e}{2\pi} x_1^2 \sqrt{n}$$

Recall that, x_1 is the fraction of the reader's signal reflected by the RFID. Hence, $x_1^2 = \frac{\text{Power of RFID's signal}}{\text{Power of Reader's signal}}$

C. RF-Cloak's Optimal Decoder and BER

After canceling the self interference and removing the random modulation, RF-Cloak's received signal is:

$$\widehat{y}(t) = h_c(t) \cdot x(t)$$

where $h_c(t)$ is the random channel from card to RF-Cloak's receiver. For each bit b, RF-Cloak receives k samples $\{Y_1, \cdots Y_k\}$ where $Y_i = H_{c_i} p_i^b$. The random channels H_{c_i} are independent and follow a complex normal distribution with zero mean and standard deviation σ. Hence, Y_i has a normal distribution with zero mean and standard deviation $\sigma_i^b = \sigma|p_i^b|$. As before the decoder will be the maximum likelihood decoder and the hypothesis test can be written as:

$$\sum_i^k \left(\frac{|Y_i|}{\sigma_i^1}\right)^2 \quad \underset{0}{\overset{1}{\lesseqgtr}} \quad \sum_i^k \left(\frac{|Y_i|}{\sigma_i^0}\right)^2$$

which for for UHF cards is:

$$|Y_{10}|^2 + |Y_{12}|^2 + |Y_{14}|^2 + |Y_{16}|^2 \quad \underset{0}{\overset{1}{\lesseqgtr}} \quad |Y_9|^2 + |Y_{11}|^2 + |Y_{13}|^2 + |Y_{15}|^2$$

And similar to before the BER will be:

$$BER = 1 - \frac{\mu^4}{(1+\mu)^4}\left(\frac{20}{(1+\mu)^3} + \frac{10}{(1+\mu)^2} + \frac{4}{1+\mu} + 1\right)$$

where $\mu = x_1^2/x_0^2$. Although, this BER equation is similar to that of the adversary, it only depends on the ratio of x_1 to x_0. Since, when the card does not reflect the reader's signal its state $x_0 \approx 0$, the BER≈ 0. In fact, even when $x_0 \leq x_1/4$, the BER is less than 0.04% which is typical for RFID communication.

9. References

[1] M. Abdelhalim, M. El-Mahallawy, M. Ayyad, and A. Elhennawy. Design and Implementation of an

Encryption Algorithm for use in RFID System. *International Journal of RFID Security and Cryptography*, 1(1), 2012.

[2] F. Adib, S. Kumar, O. Aryan, and D. Katabi. Interference Alignment by Motion. In *ACM MOBICOM*, 2013.

[3] O. Al-Rabado and G. Pedersen. Directional Space-Time Modulation: A Novel Approach for Secured Wireless Communication. In *IEEE international Conference on Communication*.

[4] Alien Technology Inc. ALN-9640 Squiggle Inlay. www.alientechnology.com.

[5] M.-S. Alouini, A. Abdi, and M. Kaveh. Sum of Gamma Variates and Performance of Wireless Communication Systems Over Nakagami-Fading Channels. *IEEE Transactions on Vehicular Technology*, 50(6), 2001.

[6] Arduino UNO Board. http://arduino.cc.

[7] U. Azad, H. Jing, and Y. Wang. Budget and Capacity Performance of Inductively Coupled Resonant Loops. *IEEE Transactions on Antennas and Propagation*, 2012.

[8] M. Barni, F. Bartolini, A. De Rosa, and A. Piva. Optimum Decoding and Detection of Multiplicative Watermarks. *IEEE Transactions on Signal Processing*, 2003.

[9] L. Batina, J. Guajardo, T. Kerins, N. Mentens, P. Tuyls, and I. Verbauwhede. Public-Key Cryptography for RFID-Tags. In *IEEE International Conference on Pervasive Computing and Communications Workshops*, 2007.

[10] S. Bono, M. Green, A. Stubblefield, A. Juels, A. Rubin, and M. Szydlo. Security analysis of a cryptographically enabled rfid device. In *USENIX Security Symposium*, 2005.

[11] M. Buettner and D. Wetherall. A Software Radio-based UHF RFID Reader for PHY/MAC Experimentation. In *IEEE International Conference on RFID*, 2011.

[12] C. Castelluccia and G. Avoine. Noisy Tags: A Pretty Good Key Exchange Protocol for RFID Tags. In *International Conference on Smart Card Research and Advanced Applications CARDIS'06*, 2006.

[13] H. Chae, D. Yeager, J. Smith, and K. Fu. Maximalist Cryptography and Computation on the WISP UHF RFID Tag. In *Conference on RFID Security*, 2007.

[14] Q. Chai and G. Gong. BUPLE: Securing Passive RFID Communication Through Physical Layer Enhancements. In *7th International Conference on RFID Security and Privacy RFIDSec'11*, 2011.

[15] J.-P. Curty, M. Declercq, C. Dehollain, and N. Joehl. *Design and Optimization of Passive UHF RFID Systems*. Springer, 2007.

[16] M. Daly. Physical Layer Encryption Using Fixed and Reconfigurable Antennas. *Ph.D. Dissertation University of Illinois at Urbana-Champaign*, 2013.

[17] DLP Design, Inc. DLP-RFID-ANT.

[18] D. Dobkin. UHF Reader Eavesdropping: Intercepting a Tag Reply. www.enigmatic-contulting.com, 2009.

[19] EPCglobal Inc. EPCglobal Class 1 Generation 2 V. 1.2.0. http://www.gs1.org/gsmp/kc/epcglobal/uhfc1g2.

[20] Eval-ADG-904R. Analog Devices.

[21] Frost & Sullivan. Global RFID Market 2011. Industry Report, 2011.

[22] F. Garcia, G. Gans, R. Muijrers, P. Rossum, R. Verdult, R. Schreur, and B. Jacobs. Dismantling MIFARE classic. ESORICS, 2008.

[23] S. Goel and R. Negi. Guaranteeing secrecy using artificial noise. *IEEE Transactions on Wireless Comm.*, 2008.

[24] A. Goldsmith. *Wireless Communications*. Cambridge University Press, 2005.

[25] S. Gollakota, H. Hassanieh, B. Ransford, D. Katabi, and K. Fu. They can hear your heartbeats: non-invasive security for implantable medical devices. In *ACM SIGCOMM*, 2011.

[26] A. Gonzalez-Ruiz, A. Ghaffarkhah, and Y. Mostofi. An Integrated Framework for Obstacle Mapping with See-Through Capabilities using Laser and Wireless Channel Measurements. *IEEE Sensors Journal*, 14(1), 2014.

[27] J. Gummeson, B. Priyantha, D. Ganesan, D. Thrasher, and P. Zhang. EnGarde: Protecting the mobile phone from malicious NFC interactions. In *MobiSys*, 2013.

[28] G. Hancke. Practical attacks on proximity identification systems. In *IEEE Symposium on Security and Privacy*, 2006.

[29] E. Haselsteiner and K. Breitfu. Security in near field communication (nfc). In *EURASIP*, 2008.

[30] J. Heiskala and J. Terry. *OFDM Wireless LANs: A Theoretical & Practical Guide*. Sams Publish., 2001.

[31] T. Hong, M. Song, and Y. Liu. Rf directional modulation technique using a switched antenna array for physical layer secure communication applications. *Progress in Electromagnetics Research*, 166, 2011.

[32] E. Inc. Universal Software Radio Peripheral. http://ettus.com.

[33] H.-S. Jang, W.-G. Lim, and J.-W. Yu. Transmit/receive isolator for UHF RFID reader with wideband balanced directional coupler. In *IEEE Microware Conference*, 2009.

[34] A. Juels, R. L. Rivest, and M. Szydlo. The blocker tag: selective blocking of rfid tags for consumer privacy. CCS'03.

[35] N. K. Kalantari, S. M. A. Ahadi, and H. Amindavar. A universally optimum decoder for multiplicative audio watermarking. In *ICME*, 2008.

[36] A. Khisti and G. Wornell. Secure transmission with multiple antennas: part II: the MIMOME wiretap channel. *IEEE Transactions on Information Theory*, 2010.

[37] H. Kortvedt and S. Mjolsnes. Eavesdropping near field communication. The Norwegian Security Conf., 2009.

[38] K. Koscher, A. Juels, V. Brajkovic, and T. Kohno. EPC RFID Tag Security Weaknesses and Defenses: Passport Cards, Enhanced Drivers Licenses, and Beyond. CCS, 2009.

[39] S. Kumar, S. Gil, D. Katabi, and D. Rus. Accurate Indoor Localization With Zero Start-up Cost. In *ACM MOBICOMM*, 2014.

[40] S. Kumar, E. Hamed, D. Katabi, and L. E. Li. LTE Radio Analytics Made Easy and Accessible. In *ACM SIGCOMM*, 2014.

[41] G. Li, D. Arnitx, R. Ebelt, U. Muehlmann, K. Witrisal, and M. Vossiek. Bandwidth Dependence of CW Ranging to UHF RFID Tags in Severe Multipath Environments. In *IEEE Conference on RFID 2011*.

[42] X. Li, J. Hwu, and E. P. Ratazzi. Using Antenna Array Redundancy and Channel Diversity for Secure Wireless Transmissions. *Journal of Communications*, 2(3), 2007.

[43] N. Marquardt and A. Taylor. RFID Reader Detector and Tilt-Sensitive RFID Tag. In *ACM CHI 2009*, 2009.

[44] Massachusetts Bay Transportation Authority. The Charlie Card Reusable Ticket System. www.mbta.com.

[45] MasterCard Worldwide. PayPass. www.paypass.com.

[46] MBTA. Reusable, rechargeable charliecards.

[47] R. Miesen, F. Kirsch, and M. Vossiek. UHF RFID Localization Based on Synthetic Apertures. *IEEE Transactions on Automation Science and Enginerring*, 10(3), 2013.

[48] R. Nandakumar, K. K. Chintalapudai, V. Padmanabhan, and R. Venkatesan. Dhwani : Secure Peer-to-Peer Acoustic NFC. In *SIGCOMM*, 2013.

[49] National Institute of Standards and Technology. Guidelines for Securing Radio Frequency Identification Systems, 2007.

[50] Netgear. A6200 USB WI-FI Adapter with Sliding Antennas. http://www.netgear.com/home/products/networking/wifi-adapters/a6200.aspx.

[51] NXP Semiconductors. MIFARE Classic. http://mifere.net/overview/mifare-standards/.

[52] K. Paget. Credit Card Fraud: The Contactless Generation. ShmooCon, 2012.

[53] J. Park and T. Lee. Channel-aware line code decision in rfid. In *IEEE Communications Letters*, 2011.

[54] J. Parsons. *The Mobile Radio Propagation Channel*. 2000.

[55] Pfizer Inc. Counterfeit Pharmaceuticals. Report, 2007.

[56] P.Nikitin et al. Effect of gen2 protocol parameters on rfid tag performance. In *IEEE RFID*, 2009.

[57] P. Pursula, M. Kiviranta, and H. Sepp. UHF RFID Reader With Reflected Power Canceller. *IEEE Microware and Wireless Components Letters*, 19, 2009.

[58] R. Ryan, Z. Anderson, and A. Cheisa. Anatomy of a Subway Hack. DEFCON, 2008.

[59] S. Sarma. Some issues related to RFID and Security, 2006. Keynote Speech in Workshop on RFID Security.

[60] O. Savry, F. Peyroula, F. Dehmas, G. Robert, and J. Reverdy. Rfid noisy reader how to prevent from eavesdropping on the communication? CHES, 2007.

[61] S. Sen, R. R. Choudhury, and S. Nelakuditi. SpinLoc: Spin Once to Know Your Location. In *ACM HotMobile*, 2012.

[62] W. Shen, P. Ning, X. He, and H. Dai. Ally friendly jamming: How to jam your enemy and maintain your own wireless connectivity. In *IEEE Symposium on Security and Privacy, Oakland*, 2013.

[63] ThingMagic. RFID Security issues - Generation2 Security. www.thingmagic.com.

[64] Travelon, Inc. RFID Blocking. www.travelonbags.com.

[65] D. Tse and P. Vishwanath. *Fundamentals of Wireless Communications*. Cambridge University Press, 2005.

[66] N. Valliappan, A. Lozano, and R. W. Heath. Antenna Subset Modulation for Secure Millimeter-Wave Wireless Communication. *IEEE Transactions on Communications*, 61(8), 2013.

[67] R. Verdult, F. Garcia, and J. Balasch. Gone in 360 secs: Hijacking with hitag2. In *Usenix Security*, 2012.

[68] J. Wang, F. Adib, R. Knepper, D. Katabi, and D. Rus. RF-compass: Robot Object Manipulation Using RFIDs. In *ACM MOBICOM*, 2013.

[69] J. Wang and D. Katabi. Dude, Where's My Card?:

RFID Positioning That Works with Multipath and Non-line of Sight. In *ACM SIGCOMM*, 2013.

[70] R. Zhou and G. Xing. nShield: A Noninvasive NFC Security System for Mobile Devices. In *MobiSys*, 2014.

[71] T. Zimmerman. Assessing the capabilities of RFID technologies. Gartner, 2009.

[72] Zipcar, Inc. www.zipcar.com/how/technology.

Relative Localization of RFID Tags using Spatial-Temporal Phase Profiling

Longfei Shangguan[1,2] Zheng Yang[2] Alex X. Liu[3] Zimu Zhou[1] Yunhao Liu[2]
[1]*Dept. of Computer Science and Engineering, Hong Kong University of Science and Technology*
[2]*School of Software and TNList, Tsinghua University*
[3]*Dept. of Computer Science and Engineering, Michigan State University, East Lansing, U.S.A.*
E-mail: {longfei, yang, zimu, yunhao}@greenorbs.com, alexliu@cse.msu.edu

Abstract

Many object localization applications need the relative locations of a set of objects as oppose to their absolute locations. Although many schemes for object localization using Radio Frequency Identification (RFID) tags have been proposed, they mostly focus on absolute object localization and are not suitable for relative object localization because of large error margins and the special hardware that they require. In this paper, we propose an approach called Spatial-Temporal Phase Profiling (STPP) to RFID based relative object localization. The basic idea of STPP is that by moving a reader over a set of tags during which the reader continuously interrogating the tags, for each tag, the reader obtains a sequence of RF phase values, which we call a phase profile, from the tag's responses over time. By analyzing the spatial-temporal dynamics in the phase profiles, STPP can calculate the spatial ordering among the tags. In comparison with prior absolute object localization schemes, STPP requires neither dedicated infrastructure nor special hardware. We implemented STPP and evaluated its performance in two real-world applications: locating misplaced books in a library and determining baggage order in an airport. The experimental results show that STPP achieves about 84% ordering accuracy for misplaced books and 95% ordering accuracy for baggage handling.

1 Introduction

1.1 Motivation

Many object localization applications need the *relative locations* of a set of objects as oppose to their *absolute locations*. The relative location of an object in a set of objects refers to the order of the object with respect to other objects along each dimension. The absolute location of an object refers to its coordinate value in each dimension. For example, in a library, to find misplaced books, we need to obtain the current order of the books on shelves rather than their absolute coordinate values.

1.2 Limitations of Prior Art

Although many schemes for object localization using Radio Frequency Identification (RFID) tags have been proposed [11, 13, 17–20, 23], they mostly focus on absolute object localization. They are not suitable for relative object localization because of two reasons. First, as the error margin achieved by most absolute object localization schemes (*e.g.*, [11, 13, 18, 23]) is still big, sorting objects based on their absolute coordinate values may not result in the correct ordering of all objects because the distance between two objects may be less than the error margin. For example, the state-of-the-art absolute object localization scheme PinIt achieves an accuracy of 16cm at the 90th percentile [18]; however, such an error margin of 16cm could allow a book to be incorrectly ordered several books away from its correct order on a bookshelf. Second, the absolute object localization schemes that can achieve small error margins require either dedicated hardware (such as USRP) [17] or multiple pre-deployed antennas as reference points [19, 20], which make them relatively harder and more expensive to deploy in practice. For example, the state-of-the-art scheme Togoram [20] can achieve millimeter localization accuracy; however, it relies on the collaboration of multiple reader antennas and requires sophisticated calibration process before putting into use.

1.3 Proposed Approach

In this paper, we propose an approach called Spatial-Temporal Phase Profiling (STPP) to RFID based relative object localization. STPP uses commercial off-the-shelf (COTS) RFID readers and passive tags and requires no pre-deployed infrastructure. The basic idea of STPP is that by carefully moving an RFID reader over a set of tags during which the reader continuously interrogating the tags, for each tag, the reader obtains a sequence of RF phase values, which we call a phase profile, from the tag's responses over time. As a reader moves closer to (or further away from) a tag, the phase value that the reader obtains from interrogating the tag changes. Thus, the phase profile of each tag corresponds to the spatial changes of the reader with respect to the tag. By analyzing the temporal dynamics in the phase profiles of a set of tags, the reader can obtain the spatial ordering among the tags. Specifically, STPP is based on the observation that *as we move the reader along a dimension in one direction, for any tag, its distance to the reader first decreases and then increases, and becomes the minimum when the reader is perpendicular above the tag along that dimension*; in other words, the distance values are symmetric around the minimum distance. Thus, in this reader moving process, if the reader continuously interrogate the tag, the phase values that reader can measure from the tag responses are also symmetric around the perpendicular point. Based on the symmetry in this observation, by moving the reader along a dimension in one direction, we can determine the order that the tags become perpendicular with the reader along that dimension, which is the order of the tags. Furthermore, by moving the reader two times, each time along a different dimension in the two dimensional space, the reader can obtain the order of the tags along each dimension. Note that an equivalent way of moving the reader while keeping the tags stationary is to move the tags altogether (with the relative positions among tags preserved) while keeps the reader stationary. For example, for airport baggage handling systems, we can keep the reader stationary while the baggages move on a conveyor belt. Therefore, our relative localization scheme can handle applications in both tag moving and antenna moving cases.

For simplicity, this paper focuses on relative object localization in a two dimensional space (*i.e.*, locating the relative order of tags on a plane). The straightforward solution to achieve this is to move the reader two times, each time along a different dimension in the two dimensional space. In this paper, we propose to achieve two dimensional object localization by moving the reader only once along any dimension. This is based on our observation that *given a sequence of objects aligned along a dimension, as we move the reader along that dimension in one direction, the larger the distance between the reader moving trajectory and that dimension, which are in parallel, the smaller the phase changes as the reader moves*. Thus, given a set of objects placed within x_1 and x_2 (where $x_1 \leq x_2$ along the X dimension) and within y_1 and y_2 (where $y_1 \leq y_2$ along the Y dimension) as shown in Figure 1, if we move the reader along the X dimension from x_1 to x_2 perpendicularly above the line from (x_1, y_2) to (x_2, y_2), objects with smaller values on the Y dimension will have smaller phase changing rate; similarly, if we move the reader along the X dimension from x_1 to x_2 perpendicularly above the line from (x_1, y_1) to (x_2, y_1), objects with larger values on the Y dimension will have smaller phase changing rate. Based on this observation, by moving the reader along the X dimension from x_1 to x_2 perpendicularly above the line from (x_1, y_2) to (x_2, y_2) (or the line from (x_1, y_1) to (x_2, y_1)), we can determine the order of the objects along the Y dimension for any point on the X dimension, in addition to obtaining the order of the objects along the X dimension; in other words, we can determine the relative location of all objects in the two dimensional region.

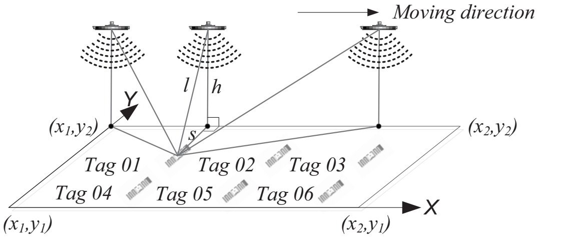


Figure 1: Illustration of STPP approach

Our STPP approach achieves relative object localization without calculating the absolute coordinate values of tags. It has two key features in comparison with prior absolute object localization schemes. First, STPP requires no dedicated infrastructure. In contrast, prior RFID based object localization schemes (*e.g.*, [11, 18]) often require dedicated infrastructure such as carefully deployed anchor tags or antennas as reference points. Second, STPP uses COTS RFID readers and tags, and requires no special hardware. In contrast, prior RFID based object localization schemes (*e.g.*, [17]) often require special hardware such as USRP.

1.4 Technical Challenges and Solutions

There are three key technical challenges in building a relative object localization system using our STPP approach. The first challenge is to achieve *high accuracy*. In STPP, phase profiles often come with noises and missing data points due to multi-path self-interference [22], which makes finding the perpendicular point for each tag challenging. To address this challenge, in this paper, we first acquire the symmetric part of each phase profile, which we call a V-zone. Within the V-zone of each phase profile, we further perform quadratic fitting on the incomplete phase values to complete the profile.

The second challenge is to achieve *high robustness*. As the mobile reader is often moved manually, the phase profile will be stretched when the movement slows down or compressed when the movement speeds up. To address this challenge, we use the Dynamic Time Warping (DTW) technique to find the V-zone within each phase profile. DTW compresses or stretches the profiles with the goal of minimizing the distance between these profiles. It naturally compensates for the warps of phase profiles and is robust to varying reader moving speed.

The third challenge is to achieve *low latency*. The time warping distance is calculated using dynamic programming algorithm in $O(MN)$ time complexity, where M and N are the lengths of a phase profile and its reference phase profile, respectively. This process can take time, especially for long phase profiles. Furthermore, as there are typically a large number of tags for localization, *e.g.*, in a library there are millions of books, detecting the V zone for each tag's profile would incur large computational overhead. To address this challenge, we perform DTW on the coarser grained representation of phase profiles. Specifically, given a phase profile with length M, we first split it into $\frac{M}{w}$ segmentations where each segment is of length w. In each segmentation, we record its maximum and minimum phase values, as well as the start and end points of this segment on the phase profile. After the segmentation, this coarser grained phase profile is used for V zone detection. Using segmentation, we thus can reduce the time complexity of DTW from $O(MN)$ down to $O(\frac{M}{w}\frac{N}{w}) = O(\frac{MN}{w^2})$.

1.5 Key Contributions

This paper represents the first study of relative object localization. Specifically, we make three key contributions in this paper. First, we propose the concept of spatial-temporal phase profiling, which can be used for RFID based relative object localization. Second, we propose algorithms to capture the spatial-temporary dynamics of RF phase profiles and algorithms to determine the tag order along each dimension. Third, we implemented STPP and evaluated its performance in two real-world applications: locating misplaced books in a library and determining the baggage order in an airport. The experimental results show that we achieve about 84% ordering accuracy for misplaced books and 95% ordering accuracy for baggage handling.

The rest of this paper proceeds as follows. In Section 2, we discuss the difficulties on relative localization and the concept of spatial-temporal phase profiling (STPP). In Section 3, we present the design details of our STPP based relative localization system. In Section 4, we present the evaluation results of our system. In Section 5, we present our findings in deploying our system in two real-world applications. In Section 6, we present the limitation and future works. In Section 7, we review related work. We conclude this paper in Section 9.

2 Spatial-Temporal Phase Profiling

In this section, we first discuss the difficulties that we experienced in our initial attempts to directly use the information that can be measured by commercial readers towards relative localization. Then, we introduce the concept of phase profiling and show how it can capture the spatial-temperal phase dynamics that helps us to achieve relative localization.

2.1 Initial Attempts

As an RFID reader sweeps over a set of tags and keeps querying them, the reader can obtain the following information that can be impacted by the changes in the spatial relationship between the tags and the readers: tag identification order, the Received Signal Strength Indication (RSSI), and the received signal phase value. We next explain the reasons that we did not use these three types of information for relative localization.

Tag Identification Order: The Class1 Generation2 (C1G2) RFID standard [4] specifies two tag identification protocols: frame slotted ALOHA [3] and tree walking [10]. Unfortunately, in both protocols, the order that the tags are identified does not correspond to the order that the reader moves across them. In frame slotted ALOHA, the identification order depends on the random numbers that tags choose by themseleves. In tree walking, the order depends on the IDs stored in the tags.

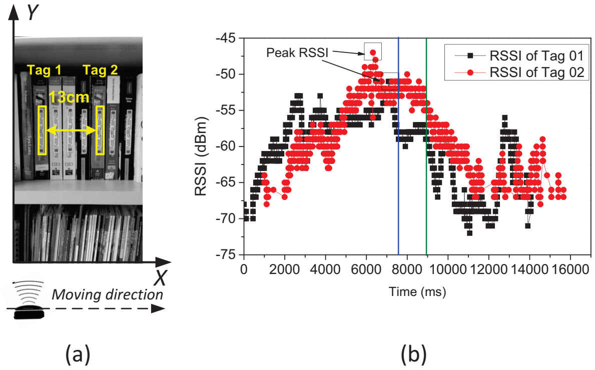


Figure 2: RSSI values measured over time for two tags

RSSI: RSSI measures the power of received radio signal, which is inverse proportional to the distance between the tag and the reader (more precisely, the reader antenna) [6]. As a reader moves across a set of tags, for each tag, the RSSI values measured by the reader should increase and then decrease because the distance between the tag to the reader first decreases and then increases; thus, by ordering the tags according to the time that their peak RSSI values appear, the reader obtains the order of the tags along the moving direction. Unfortunately, this works only in theory because of the multiple paths that the signal traverses. To evaluate the multi-path impact, we conducted an experiment by attaching tags to the books on a shelf and moving the reader from left to right as shown in Figure 2(a). Figure 2(b) shows the RSSI values that the reader measures over time for two tags labeled 01 and 02, where tag 01 is placed 13cm to the left of tag 02. The left and right vertical lines corresponds to the time that the reader passes through tag 01 and 02, respectively. From this figure, we first observe that for both tags, their RSSI values fluctuate and their peak RSSI values appear before the reader moves across them. Second, the order of the two tags based on the time that their peak RSSI values appear is inconsistent with the actual tag order.

RF Phase Values: Phase is a basic attribute of a signal along with amplitude and frequency. The phase value of an RF signal describes the degree that the received signal is offset from sent signal, ranging from 0 to 360 degrees. Let l be the distance between the reader antenna and the tag, the signal traverses a round-trip ($2l$) in each backscatter communication. Apart from the RF phase rotation over the distance, both the antenna and the tag will introduce additional phase distortion. Specifically, let θ_{Tx}, θ_{TAG}, and θ_{Rx} be the phase rotation introduced by the reader's transmission circuit, the tag's reflection characteristic, and the reader's receiver circuits, respectively. The phase measurement θ output by the reader thus can be expressed as:

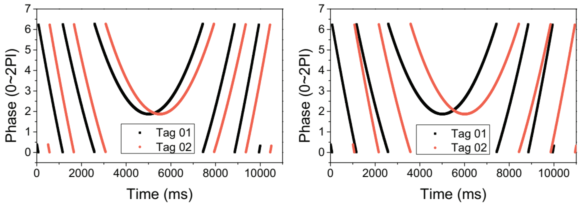
$$\begin{cases} \theta = (2\pi\frac{2l}{\lambda} + \mu) \mod 2\pi \\ \mu = \theta_{Tx} + \theta_{Rx} + \theta_{TAG} \end{cases} \quad (1)$$

where λ is the wavelength, μ is system noise. Most commercial RFID readers (such as ImpinJ R420 [1]) are able to report θ as the difference of the transmitted and the received signal. Given the ultra-high working frequency of the commercial passive RFID system, it is possible to achieve mm-level ranging accuracy in theory [20]. However, as the phase is a periodic function that repeats every λ in the distance of signal propagation, we cannot use phase value to pinpoint relative tag locations.

2.2 Phase Profile

The basic idea of our approach is that by carefully moving an RFID reader over a set of tags during which the reader continuously interrogating the tags, for each tag, the reader obtains a sequence of RF phase values, which we call a *phase profile*, from the tag's responses over time. Considering Figure 1 where the set of tags are placed within x_1 and x_2 along the X dimension and within y_1 and y_2 along the Y dimension, suppose we move the reader along the X dimension from x_1 to x_2 perpendicularly above the line from (x_1, y_2) to (x_2, y_2). Taking tag 01 as an example, its distance to the reader first decreases until the reader is perpendicular above tag 01, and then increases. According to Equation 1, the phase of the received signal will also decrease first and then increase. Since the range of any phase value is $[0, 2\pi)$, when this phase value decreases to 0, it immediately jumps to 2π. This process repeats until the reader reaches the perpendicular point right above tag 01, where the received phase stops decreasing and starts to increase from a certain value within $[0, 2\pi)$; when the phase value increases to 2π, it will immediately drop to 0 and then increases again. Such periodic change of phase values is reflected visually as follows: (1) The phase profile of each tag has a "V-zone" where its bottom occurs at the time when the reader is perpendicular above the tag. (2) Multiple curves are symmetrically distributed on both sides of the V-zone where each curve except the V-zone spans the whole range of $[0, 2\pi)$. A curve is called one *period* of the phase profile.

Given a layout of tags and the reader, their relative positions and the reader moving speed, assuming the speed is steady, we can calculate the phase profile of each tag, which we call the *reference phase profile*. Consider tags 01 and 02 and the reader in Figure 1, and suppose the reader moves at a constant speed of 0.1m/s along the line from x_1 to x_2 perpendicularly above the line from (x_1, y_2) to (x_2, y_2). Suppose the distance between x_1 and x_2, the height of the reader, and the distance from tag 02 to the line from (x_1, y_2) to (x_2, y_2) are 3m, 1m and 0.5m, respectively. Figure 3(a) shows the reference phase profiles of tags 01 and 02 when their distance is 5cm. This figure shows that the phase profiles of tag 01 and tag 02 have similar V-zone patterns.

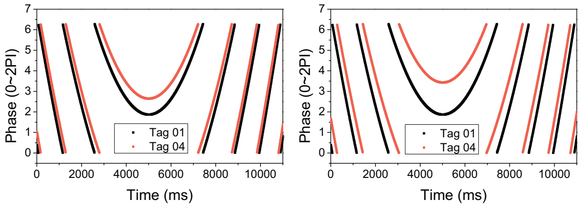


(a) X dimension spacing = 5cm (b) X dimension spacing = 10cm

Figure 3: Reference phase profile along X-axis

Given the phase profiles of multiple tags, the order that the reader passes through the tags along the X-axis is consistent with the order that the V-zones reach their bottom. By ordering the V-zones according to the time that they reach their bottoms, we can order the tags along the X-axis. Figure 3(a) shows that the V-zone of tag 01 reaches its bottom earlier than that of tag 02, which is consistent with the order that the reader passes through the tags. Furthermore, the longer the distance between two adjacent tags, the longer the time duration between the bottoms of two V-zones is. For example, Figure 3(b) shows the reference phase profiles of tags 01 and 02 when their distance is 10cm. As we increase the distance between the two tags from 5cm to 10cm, the time duration between the two V-zones also increases.

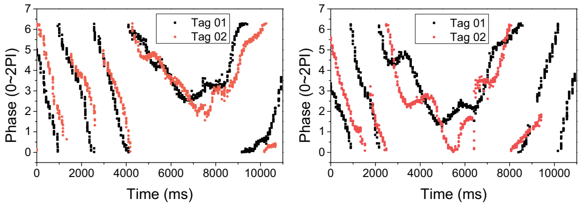
Given the phase profiles of multiple tags, the larger the bottom phase value of a V-zone is, the longer the distance between the tag that corresponds to the V-zone and the reader. By ordering the V-zones according to the phase value of their bottoms, we can order the tags along the Y-axis. Figure 4(a) shows that the V-zone bottom phase value of tag 04 is smaller than that of tag 01, which means that tag 04 is farther away than tag 01 with respect to the reader. Furthermore, the larger the two bottom phase values of two V-zones differ, the larger the distance between the two corresponding tags along the Y-axis. Figure 4(a) and (b) shows the phase profiles of tag 01 and 04, whose distances along the Y-axis are 5cm and 10cm, respectively. We observe that by increasing the tag distances from 5cm to 10cm, the distances between the bottom phase values of the two corresponding V-zones increases.



(a) Y dimension spacing = 5cm (b) Y dimension spacing = 10cm

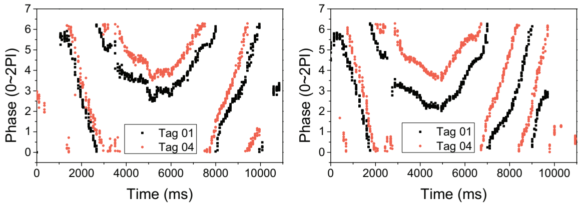
Figure 4: Reference phase profile along Y-axis

To validate the above observations from reference phase profiles, we reproduce the layout of tags in Figure 1 on a white board. We attach an RFID reader on a shopping cart and wheel the cart along the X-axis in the positive direction. The speed of the cart is also set to be 0.1m/s. Figure 5 and Figure 6 shows the two measured phase profiles. From these figures, we can derive the same observations as above. Besides, we also found that due to channel instability, the phase profiles outside the V-zone are fragmentary. It is thus error-prone to connect the whole profile into a big V-zone for tag ordering.



(a) X dimension spacing = 5cm (b) X dimension spacing = 10cm

Figure 5: Measured phase profile along X-axis



(a) Y dimension spacing = 5cm (b) Y dimension spacing = 10cm

Figure 6: Measured phase profile along Y-axis

3 System Design

In this section, we present the details of our STPP approach to obtain the order of the tags along the X- and Y-axis, respectively. Without loss of generality, we assume that the reader moves along the X-axis from left to right.

3.1 Tag Ordering along X-axis

The profile segment within the V-zone differs from the other parts of the phase profile from two aspects. First, it changes continuously without jumping from 0 to 2π. Second, it is self-symmetric around the time point that the reader is perpendicular with the tag, which we call the perpendicular point. A straightforward solution to detect the V-zone is to use a sliding window to find the profile segment that satisfies these two properties. However, in reality, due to multi-path self-interference, the phase profile often has missing values within the V-zone as shown in Figure 6(a). Thus, this solution is unreliable for V-zone detection.

3.1.1 Detecting V-zone with Time Warping

Our basic approach is to match the measured phase profile against a pre-calculated reference phase profile, and try to find where the V-zone appears in the measured phase profile. As the reader is often hand held and moved manually, the phase profile become stretched when the movement slows down and compressed when the movement speeds up during the movement. Thus, subsequence matching algorithms (such as the KMP algorithm [7]) will not work for our V-zone detection. To find the place where the V-zone appears, we need to stretch or compress the calculated profile to match the corresponding V-zone on the given phase profile.

To address this issue, we use the Dynamic Time Warping (DTW) technique to match the V-zone in the calculated phase profile against the measured phase profile. DTW is a transformation that automatically compresses or stretches a sequence with the goal of minimizing the distance between these sequences. It naturally compensates for the shifts among different phase profiles caused by the varying reader moving speed. The input to the DTW algorithm consists of a reference phase profile P of length N and a measured phase profile Q of length M. DTW first constructs a distance matrix $D_{M\times N}$ where each element $D_{i,j}$ is defined as the Euclidean distance between p_i and q_j:

$$D_{i,j} = \|p_i - q_j\|$$

where p_i and q_j are the i^{th} and j^{th} elements of the phase profiles P and Q, respectively. The output of the DTW algorithm is a warping path $\mathscr{L} = \{l_1, l_2, ..., l_k\}$ such that the total cost $C_{\mathscr{L}}$ of the warping path L is minimized:

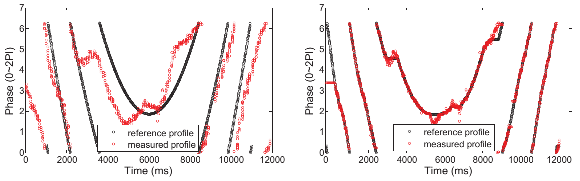
$$\underset{\mathscr{L}}{\operatorname{argmin}} \quad C_{\mathscr{L}} = \sum_{i=1}^{k} D_{x(l_i),y(l_i)}$$

where $l_i = (x,y) \in [1:M] \times [1:N]$ for $l \in [1:k]$.

To generate the optimal warping path, DTW constructs the cost matrix $C_{i,j}$ using dynamic programming. The optimal substructure is defined as:

$$C_{i,j} = D_{i,j} + min\ \{C_{i,j-1}, C_{i-1,j}, C_{i-1,j-1}\}$$

Figure 7(b) shows the matching result using DTW. It shows that the V-zone of the measured profile matches well with that of the reference profile. On the reference profile, as the start and the end point of the V-zone is known a priori, it is easy to locate the corresponding V-zone on the measured profile.



(a) Before warping

(b) After warping

Figure 7: V-zone detection using DTW

3.1.2 Optimizing V-zone Detection Efficiency

The core of DTW is dynamic programming whose complexity is $O(NM)$. This process may take some time because the phase profiles may be long (*e.g.*, typically around 400 samples) and the number of tags may be large. To improve efficiency, we apply DTW on the coarser grained representations of phase profiles. Given a phase profile P, we split it into d segments: $S_P = \{s_{P,1}, s_{P,2}, ..., s_{P,d}\}$. For each segment $s_{P,i}$, we further record its segment range $s^R_{P,i}$ and time interval $s^T_{P,i}$. Formally, the segment range $s^R_{P,i}$ is defined as:

$$\begin{aligned} s^R_{P,i} &= \{s^L_{P,i}, s^U_{P,i}\} \\ s^L_{P,i} &= min\ \{p_a, ..., p_b\}, \quad s^U_{P,i} = max\ \{p_a, ..., p_b\} \end{aligned}$$

where $s^L_{P,i}$ and $s^U_{P,i}$ are the minimum and maximum phase values within i^{th} segment. a and b are the begin and the end index of the phase profile within this segment. Note that if within a segment the phase value jumps from 0 to 2π, we split the segment into two segments at that point so that no segment contains such phase value jumping. Figure 8 shows an example segmentation. In this figure, we represent the original profile with 25 segments, with each consists of its segment range and time interval.

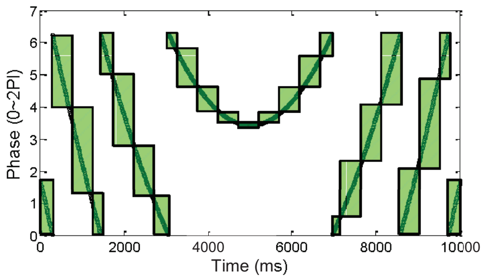


Figure 8: Phase profile segmentation;

Given two phase profiles P and Q, we first acquire their segmented presentation S_P and S_Q, with each contained J and K segments, respectively. Similar to DTW, we construct a distance matrix $D_{J \times K}$, where each element $D_{i,j}$ is defined as the distance between the segmentation $s_{P,i}$ and $s_{Q,j}$. It is intuitively the distance of their two closest points:

$$D_{i,j} = \begin{cases} \|s^L_{P,i} - s^U_{Q,j}\|, & if\ (s^L_{P,j} > s^U_{Q,j}) \\ \|s^L_{Q,j} - s^U_{P,i}\|, & if\ (s^L_{Q,i} > s^U_{P,i}) \\ 0, & otherwise \end{cases}$$

After compute each element in the matrix $D_{J \times K}$, we align S_P and S_Q using dynamic programming. The optimal substructure defined as follows:

$$C_{i,j} = min\ \{s^T_{P,i}, s^T_{Q,j}\} \cdot D_{i,j} + min\ \{C_{i,j-1}, C_{i-1,j}, C_{i-1,j-1}\}$$

Using segmentation, we reduce the time complexity of DTW from $O(MN)$ down to $O(\frac{M}{w}\frac{N}{w}) = O(\frac{MN}{w^2})$ where w is the length of each segment. We need to choose the value for w carefully to tradeoff between efficiency and accuracy. The larger the w is, the more efficient DTW is, but the less accurate our V-zone detection is due to the unclear outline of the segmented phase profile. In Section 4, we investigate how to select a proper w value.

After we detect the V-zone for a tag in its phase profile, we search for the time point with the smallest phase value within the V-zone. However, due to the multi-path self-interference, the measured phase profile often contains noise and missing values, which may cause the nadir of the V-zone profile to wrap around. In this work, we use the quadratic fitting technique to minimize such influences. Once the fitting function is determined, by referring the time point when the fitting function achieves the minimum value, we sort this tag together with those tags whose V-zones have already been determined. Figure 9 shows a concrete example. In this example, three tags are attached on a white board, then the antenna moves along the X-axis from the right to the left at a speed of approximate 0.1m/s. The distance between tag 03 and tag 01, tag 01 and tag 02 are 15cm and 2cm, respectively. After performing the quadratic fitting on these phase profiles, we see a clear lag between the phase profiles of these three tags. Based on the time point when the fitting function achieves the minimum value, we further determine the order of these three tags as 01, 02, and 03, which is coherent with the actual order.

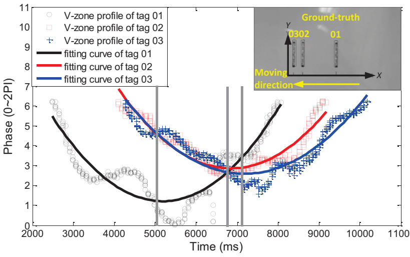


Figure 9: Tag ordering with quadratic fitting

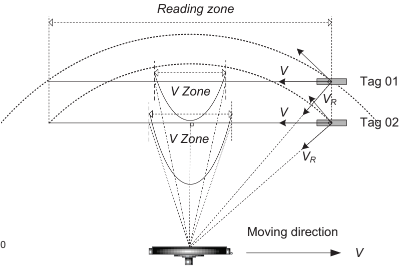


Figure 10: Reader movement model

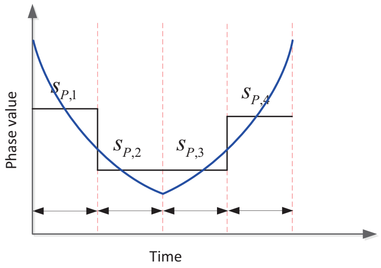


Figure 11: Examples of coarse representation of V-zone profile

3.2 Tag Ordering along Y-axis

The movement model of the reader when it passes by two tags at a constant speed v is shown in Figure 10. Intuitively, the radial velocity v_R of the tag is inverse proportional to its distance with respect to the antenna. That is, the larger the distance between the tag and the moving trajectory of the reader, the lower the radial velocity of this tag. The lower radial velocity further leads to a smaller phase changing rate, therefore a shallower V-zone profile. Based on the above observation, we propose another segmentation based method to determine the tag order along the Y-axis.

3.2.1 Tag Ordering via V-zone Profile Comparison

The basic idea to determine tag ordering along the Y-axis is to comparing their phase changing rates. One straightforward method is to first derive the span and offset of the quadratic model, and then uses these two parameters to calculate the phase changing rate. However, in reality, if the tags are placed close to each other (such as 5cm), the V-zone profiles of these tags would be similar and would lead to similar curve fitting results. In STPP, we compare the phase changing rate by jointly considering multiple local phase profile segments within the V-zone profile. Notice that the V-zone profile may vary in length due to the random access property of ALOHA protocol [3]. Thus, we first split each profile into equal number of segments to facilitate the comparison. Within each segment of the V-zone profile, we calculate the mean value of phase values. Therefore, given a phase profile P, we can get its coarse representation by using the set of mean values, *i.e.*, we represent the V-zone profile P by $S(P) = \{s_{P,1}, s_{P,2}, ..., s_{P,k}\}$, where k is the number of segments and $s_{P,k}$ is the mean value of k^{th} segment. Averaging over all phase values within each segment will eliminate the impact of noise introduced in phase value measurements. Since each segment corresponds to one specific time window, the average phase value also reflects the accumulated phase changing rate within each segment. By calculating the average phase values, we can improve the robustness of our scheme. Figure 11 shows an example coarse representation of the V-zone profile. In this figure, the phase value within each segment is represented by its mean value.

To determine the order of two tags along the Y-axis, we compare the coarse representation of their V-zone profiles, say $S(P)$ and $S(Q)$, using the following metric:

$$O(P,Q) \quad = \quad \sum_{i=1}^{k} \lceil \frac{s_{P,i} - s_{Q,i}}{s_{P,i}} \rceil$$

Generally, if the phase changing rate of P is smaller than that of Q, for each segment i, $s_{P,i}$ will be larger than $s_{Q,i}$. Therefore, $O(P,Q)$ will be close to k. On the contrary, if the phase changing rate of P is larger than that of Q, $s_{P,i}$ will be no larger than $s_{Q,i}$. Here $O(P,Q)$ will be close to 0 accordingly. Therefore, we can determine the tag order along the Y-axis based on the value of $O(P,Q)$.

3.2.2 Optimizing the Ordering Efficiency

The core of determining the tag order along the Y-axis is to compare the V-zone profiles by using the metric $O(P,Q)$. This process may take some time because we need to compare each pair of phase profiles. For example, it takes $\frac{M(M-1)}{2}$ comparison to determine the order of M tags along the Y-axis. To speed up this process, we further introduce a new metric $G(P,Q)$ to measure the gap between two phase profiles P and Q. It is defined as follows:

$$G(P,Q) = \sum_{i=1}^{k} \|s_{P,i} - s_{Q,i}\|$$

where $\|s_{P,i} - s_{Q,i}\|$ is the Euclidean distance between the mean phase value $s_{P,i}$ and $s_{Q,i}$. In an intuitive level, $G(P,Q)$ is proportional to the physical spacing of these two tags. *i.e.*, the larger the physical spacing between these two tags, the larger the $G(P,Q)$ will be. For M tags, we then randomly choose one tag as the pivot. Let P be the V-zone profile of this pivot, then we calculate $O(P,Q)$ and $G(P,Q)$ between P and each profile Q of the remaining tags. By doing so, we can not only determine the relative order between the pivot tag and other tags, but also acquire the relative distance of these tags. Therefore, we can order these M tags with only $M-1$ comparison, which is significantly smaller than $\frac{M(M-1)}{2}$.

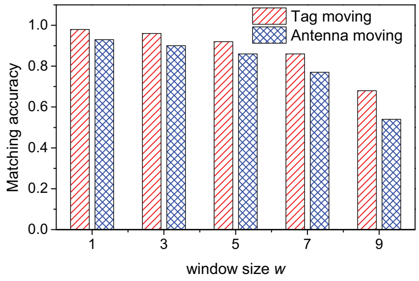


Figure 12: Window size vs. accuracy;

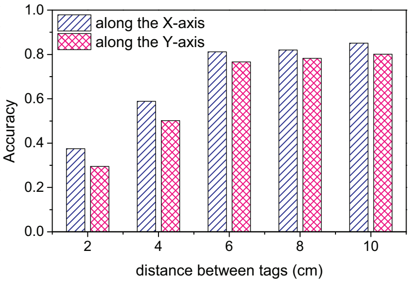


Figure 13: Tag moving case

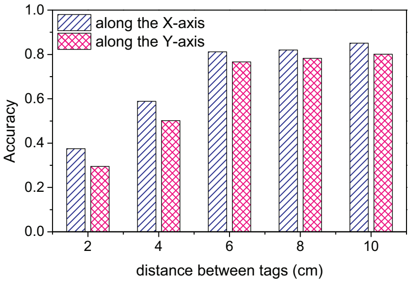


Figure 14: Antenna moving case

		Tag population size within a reading zone					
		n=5	n=10	n=15	n=20	n=25	n=30
Tag moving case	along X-axis	0.963	0.954	0.952	0.937	0.906	0.884
	along Y-axis	0.917	0.903	0.878	0.874	0.863	0.856
Antenna moving case	along X-axis	0.873	0.865	0.861	0.852	0.841	0.813
	along Y-axis	0.809	0.806	0.798	0.779	0.765	0.754

Table 1: Tag population vs. ordering accuracy

4 System Evaluation

4.1 Implementation

Hardware: Our system consists of a COTS UHF RFID reader, a directional antenna, and a set of passive tags. To account for device diversity, we have tested our system using different hardware, including an ImpinJ R420 reader, an ImpinJ Threshold RFID Antenna IPJ-A0311, an Alien ALR-8696-C antenna, and four types of passive tags: Alien ALR-9610, ALN-9662, ALN-9634, and ALN-9720. For diversity, we choose four types of tags of different size and shape.

Software: We implemented our algorithms in Java, which were executed on a Lenovo PC equipped with an Intel(R) Celeron G530 CPU and 4G RAM. The PC is connected to the RFID reader via Ethernet. The reader is programmed to continuously query the RFID tags on the 6^{th} channel in the 920 $\sim$ 926 MHz ISM band and returns the signal phase for each tag reply.

4.2 Deployment

One deployment issue is to determine the number of periods that the reference phase profile should contain. In theory, the reference phase profile should contain the same number of periods as the measured profile. In order to obtain a proper reference phase profile, we put the reader 30cm (a common distance between a librarian and a bookshelf) away from the tags. We collected phase profiles by holding the reader and passing 200 tags for 15 times. Of the 3,000 phase profiles that we collected, more than 97% of them contain 4 partial or complete periods. Thus, we generate a 4-period reference phase profile as the default setting in our experiment.

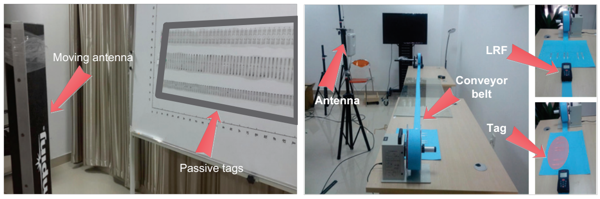
Another deployment issue is to determine the height that the antenna should be moving across the tags. As STPP uses the phase changing rate of each tag to determine its relative order along the Y-axis, we need to place the antenna at a height such that the tags with different Y coordinates differ in phase changing rate. This can be ensured if all the tags are either above or below the antenna along the Y-axis since their antenna to tag distances would differ from each other. For example, in library, we can put the antenna at the bottom of the lowest shelf so that each tag has a different distance to the reader, which is moving along the X-axis. In our experiments, we simply place the antenna at a height below all tags.

4.3 Micro-Benchmarks

Experimental setup: We have two experimental cases: the antenna moving case and the tags moving case. In the antenna moving case, we partition 150 tags into 3 groups and attach them on a white board as shown in Figure 15(a). The antenna is fixed on a wheeled chair which is pushed manually at a rough speed of 0.3m/s. This experimental setup simulates the misplaced book locating application in libraries where a librarian moves a reader across a bookshelf.

In the tag moving case, we use a conveyor belt and a tape to compose a mobile RFID system as shown in Figure 15(b). The antenna is placed 1m away from the tape and 1m above the top of the winder. We attach a set of tags on the tape, which move at a constant speed of 0.3m/s. This case simulates the baggage handling application in airports where baggage or cargos attached with RFID tags are delivered on a conveyor belt.

Evaluation Metrics: We mainly use the metric of ordering accuracy defined in Equation 2. A tag is ordered incorrectly in a sequence of tags if and only if the detected order of the tag is not equal to the actual order of



(a) Antenna moving case (b) Tag moving case

Figure 15: Experimental setup

that tag. For example, suppose there are five tags and the correct order of these five tags is 1-2-3-4-5. If the output of our scheme is 1-2-4-3-5, then we immediately know that the tag 4 and tag 3 are ordered incorrectly, and thus the accuracy is 3/5=60%.

$$Ordering\ Accuracy = \frac{\#\ of\ tags\ ordered\ correctly}{\#\ of\ tags\ in\ total} \quad (2)$$

Determining a proper window size w: In general, a larger window size contributes to higher efficiency but lower accuracy. As shown in Figure 12, the ordering accuracy of STPP remains high for small window sizes (*e.g.*, nearly 98% when $w = 3$), decreases slightly with window sizes increased from 3 to 5, and drops sharply for window sizes larger than 5. Therefore we set w to be 5 in our experiments to tradeoff between latency and accuracy.

Tag-to-tag distance vs. Ordering accuracy: As shown in Figure 13, when each tag pair is placed very close (*e.g.*, 2cm apart), STPP achieves an ordering accuracy of only 42% along the X-axis and 23% along the Y-axis in the tag moving case. The ordering accuracy then increases dramatically as we slightly increase the tag-to-tag distance: 92% and 88% along the X-axis and the Y-axis respectively for tag-to-tag distance of 10cm. The similar trend is observed for the antenna moving case as shown in Figure 14 where the ordering accuracy remains high for tag-to-tag distances larger than 8cm.

Tag population vs. Ordering accuracy: Commercial RFID reader have limited reading rate. If the reading zone of the antenna contains a large number of tags, we will have under-sampling of phase readings which potentially degrades the ordering accuracy. We change the tag populations from 5 to 30 within the reading zone of the antenna and examine the performance of STPP. The distance between two adjacent tags is randomly chosen in the range of $[2cm, 10cm]$. We present the experimental results in Table 1 to compare the data values. As shown in this table, when the tag population is small within the reading zone of the antenna, *e.g.*, $n = 5$, STPP achieves satisfactory performance, with ordering accuracies of above 90% and 80% for the tag moving and antenna moving cases, respectively. As we steadily increase the tag population within the reading zone, the ordering accuracy degrades gradually in both two cases. When the tag population reaches 30, the ordering accuracy remains at an acceptable level, with average accuracies of above 0.85 and 0.75 for tag and antenna moving cases, respectively. This result indicates that the performance of STPP will degrade a little bit when the tag population increases.

4.4 Macro-Benchmarks

We evaluated STPP in comparison with the following four schemes that are implementable on COTS RFID readers:

1. G-RSSI: This is a straightforward scheme that uses RSSI value changes to infer tag orders along the X-axis.

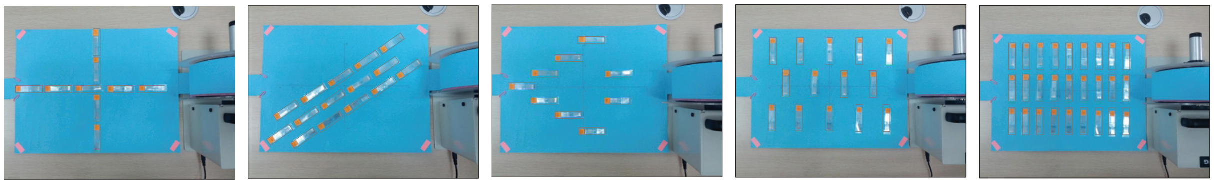
2. OTrack [16]: This scheme combines RSSI dynamics and tag successful reading rates to determine tag orders along the X-axis.

3. Landmarc [13]: This scheme uses multiple reference tags to calculate the absolute location of a tag in 2 dimensional region.

4. BackPos [11]: This scheme uses RF phase values and the hyperbolic positioning technique to calculate the absolute location of a tag in 2 dimensional region.

Our experimental results show that STPP significantly outperforms the other four schemes for the accuracy of relative localization. We compare the ordering accuracy of these schemes under various layout settings as shown in Figure 16. In each setting, we repeat the experiment 100 times and use their average ordering accuracy values. The distance between adjacent tags ranges from 1cm to 10cm. As shown in Figure 17, G-RSSI and Landmarc achieve similar low ordering accuracy values of below 25% along both axes. Using both RSSI dynamics and tag successful reading rates, OTrack outperforms G-RSSI and Landmarc, yet can only reach an ordering accuracy of below 50%, which is too low for real-world applications. With more precise signal measurement, BackPos can locate each tag and further distinguish their relative order with an average ordering accuracy of 80%. In contrast, STPP achieves an average ordering accuracy of more than 88%.

Our experimental results show that STPP scales better than the other four schemes as adjacent tag distance decreases. To perform this evaluation, we choose a population of 20 tags and vary the adjacent tag distance from 100cm to 10cm. Figure 18 shows the box plot of the accuracy values of different schemes as we vary the distance. The whisker indicates values outside the upper and lower quartiles. From this figure, we can observe



(a) Test case 1 (b) Test case 2 (c) Test case 3 (d) Test case 4 (e) Test case 5

Figure 16: Tag layout settings

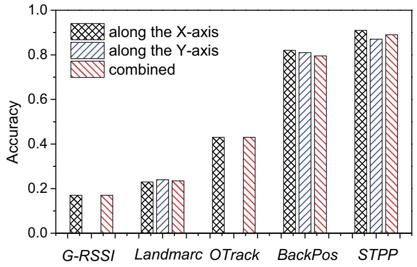


Figure 17: Accuracy vs. schemes

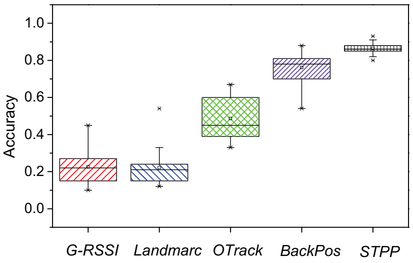


Figure 18: Accuracy vs. tag distance

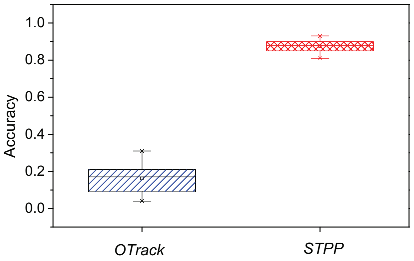


Figure 19: Accuracy vs. population

that the median accuracy of STPP is significantly higher than that of other four schemes. Besides, the likely range of variation (IQR) of STPP is the smallest as the adjacent tag distance decreases.

Our experimental results show that STPP scales better than the other four schemes as tag population size increases. To perform this evaluation, we choose 10cm to be the adjacent tag distance and vary the tag population from 5 to 30. As G-RSSI, Landmarc, and BackPos are insensitive to tag population sizes, we thus compare STPP with OTrack. Figure 19 shows the box plot of the accuracy values of different schemes as we vary the tag population size. From this figure, we observe that likely range of variation (IQR) of STPP is significantly smaller than that of OTrack.

5 Case Studies

We deployed our STPP based relative RFID tag localization system in two real-world applications: a misplaced book locating system in a library and a baggage handling system in an airport. In this section, we present our experimental results with these two case studies. Note that our relative localization scheme is not limited to these two applications. Other applications (such as locating suspicious baggage and warehouse stocktaking) can also benefit from our localization scheme.

5.1 Misplaced Book Locating in Library

A major task for librarians is to locate misplaced books and relocate them to the right place. Note that library books are typically strictly ordered based on their IDs so that borrowers can find a specific book easily. To help locate misplaced books, we deploy our STPP system in a school library. For one bookshelf in the library, we attach 90 RFID tags to 90 books, one tag per book. These books are placed on three levels. The thickness of each book spans from 3cm to 8cm. We attach an RFID antenna on a cart and manually push it across the bookshelf from left to right, as shown in Figure 20. Here we simply put the antenna at the height

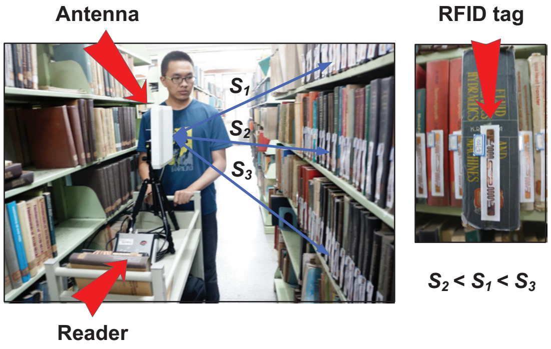


Figure 20: Locaking misplaced books

This case study also shows that STTP can achieve high relative localization accuracy. We sweep these 90 books over 50 times. The result shows that our relative localization scheme achieves an accuracy of 0.84 on average. This implies that in most cases, STPP can precisely pinpoint the relative location of the misplaced book. For the remaining cases, although STPP cannot correctly find the relative location of tags, it still helps the librarian to narrow down the searching space. Figure 21 shows the order of the books that we obtained in one experiment, whee each dot represents a book and each cross represents a book that we ordered incorrectly. Note that the gap between two dots reflects the distance between two tags.

From this figure, we observe that all incorrectly ordered books are those thin ones as their tags are much closer.

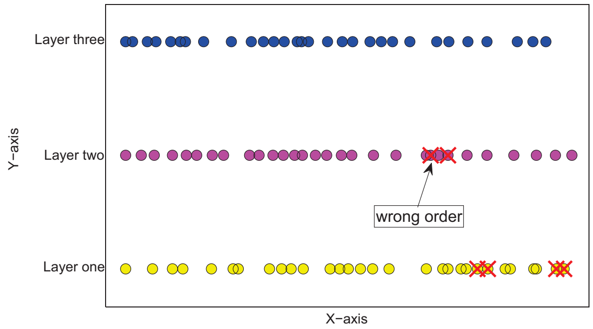


Figure 21: Layout of detected books by STPP

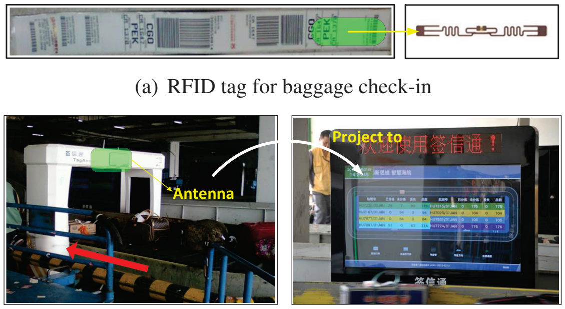
We also conducted experiments to evaluate the ability of STPP in detecting misplaced books. We randomly picked one book, two books, and three books from a bookshelf and inserted them into a differently chosen location on this bookshelf. This location is randomly chosen from the range of 2 books away from the original place to 10 books away. Each case was repeated 100 times. The detection success rate is shown in Table 2.

	Detection success rate
1 book	98%
2 books	97%
3 books	98%

Table 2: Result of misplaced book detection by STPP

5.2 Baggage Handling in Airport

To avoid mis-delivered baggages, baggage handling systems in airports need to find the order of the baggages on the conveyor belt [2]. Although the size of one baggage item is usually large, the distance between adjacent tags (attached to different baggages) can be rather close due to the arbitrary orientation of baggage on the convey belt. It is thus critical to pinpoint the relative order of baggage with high resolution. We deployed our STPP system at Terminal One, Sanya Phoenix airport, Sanya, Hainan Province, China. Three RFID reader antennas are deployed at three places on the tunnel as shown in the left figure in Figure 22(b). Based on the tag ordering information, the visualization module displays each baggage and tracks its movement on the baggage conveyor belt, as shown in the right figure in Figure 22(b). As reference tags and antennas, which are the essential part of the localization scheme Landmarc and BackPos, cannot be deployed on the commercial baggage handling system, we thus compare STPP with OTrack and G-RSSI in this case study. Our experiments were carried out during three periods: 7:00AM∼9:00AM, 13:00PM∼15:00PM, and 19:00PM∼21:00PM, during which over 1,000 pieces of baggage from 9 flights are handled.



(a) RFID tag for baggage check-in

(b) Baggage handling in Terminal One, Sanya Phoenix airport

Figure 22: Baggage handling in the airport

This case study shows that STTP can achieve high relative localization accuracy. Table 3 shows the accuracy results of STPP in comparison with G-RSSI and OTrack during the three time periods. During the peak hours of 7:00AM∼9:00AM and 19:00PM∼ 21:00PM, during which the distance between each baggage is typically smaller than 20cm, our STPP achieves accuracy values of 97% and 96%, respectively; whereas OTrack achieves an accuracy of 88% for both time periods and G-RSSI achieves accuracy values of 59% and 51%, respectively. During the off peak hours of 13:00PM∼15:00PM, our STPP, OTrack, and G-RSSI achieve accuracy values of 97%, 95%, and 72%, respectively.

	7:00∼9:00	13:00∼15:00	19:00∼21:00
STPP	388/400=97%	224/230=97%	422/440=96%
OTrack	352/400=88%	218/230=95%	388/440=88%
G-RSSI	234/400=59%	166/230=72%	226/440=51%

Table 3: Accuracy of STPP, OTrack, and G-RSSI

We further examine the ordering latency of OTrack and STPP. In this trial of experiments, we use OTrack and STPP to detect the order of 100 baggages on a moving conveyor. The CDF of the ordering latency incurred by each scheme is shown in Figure 23. As the result indicates, the average latency of STPP is 1.473s, which is slightly hight than that of OTrack.

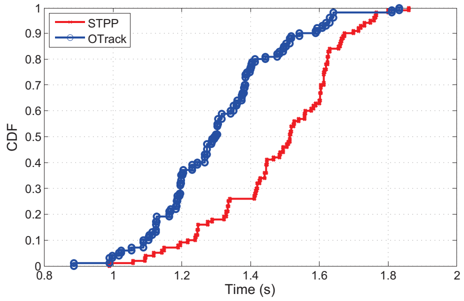


Figure 23: Ordering latency of STPP and OTrack

6 Limitation and Future Works

Improving accuracy: Our accuracy still has room to improve. One possible direction is to sweep tags multiple times and average their results. In future, we plan to leverage the advanced signal processing techniques to minimize the phase noises and exploit the geometry relationship among tags to improve the localization accuracy.

Enhancing robustness: Currently we require the reader to move along a straight line that crosses the targeting items. However, the line may not be strictly straight in practice. In future, we plan to model the impact of irregular reader motions on the phase profile, and enhance the robustness of our relative localization scheme by filtering out phase values introduce by irregular reader motions.

Extending to 3-Dimensional space: Currently we focus on the relative tag localization in a 2D space. A straightforward approach to handle the 3D space is to move the reader three times, each time along a different dimension in the 3D space. Thus, the reader can obtain the order of the tags along each dimension. In future, we plan to study ways to extend our spatial-temporal phase profiling approach for 3D relative tag localization.

7 Related Work

RSSI based approach: Early RF-based localization schemes primarily rely on RSSI information to acquire the absolute location of an object [13, 16, 21, 23]. They typically pre-deploy tags densely on a monitoring region as anchors, and then use the RSSI values of these anchor tags as references to locate a specific tag [13, 23]. Succeeding works explore the anchor-free approach by either modeling the signal propagation process in complex environment [21] or taking a combination of various signal features (*e.g.* the RSSI and the tag's reading rate [16]). The major limitation of RSSI-based approaches is that they are highly sensitive to multi-path propagation, and thus difficult to achieve high-precision localization. Furthermore, RSSI is also impacted by antenna gain [8], which adds uncertainty to localization accuracy.

Phase based approach: There is a growing interest in using phase values to estimate the absolute location of an object. Pioneer work uses hyperbolic localization techniques [11, 19] or Angle of Arrival (AoA) information [5,9,14] to locate tags by measuring the phase difference between the received signals at different antennas. To reduce the hardware deployment cost, state-of-the-art systems use synthetic aperture radar (SAR) to simulate multiple antennas to extract RF information [15, 18]. For instance, by leveraging antenna motion, PinIt achieves a location accuracy on the order of centimeters [18]. Another line of work employs multiple antennas to construct a hologram for tag localization [12, 20].

Our work is inspired by the above works in phase-based tag localization, but we focus on leveraging reader mobility to generate phase profiles for tag localization. In this setting, PinIt [18] is perhaps most related to ours. It locates RFID tags by analyzing their multi-path profiles collected by a moving antenna. However, the intuition behind PinIt is that nearby RFID tags experience a similar multi-path environment and thus exhibit similar multi-path profiles. In contrast, the intuition behind our scheme is that by analyzing the spatial-temporal dynamics in the phase profiles of a set of tags, we can calculate the spatial ordering among tags. Moreover, PinIt relies on dedicated hardware (*i.e.*, USRP) to capture the multi-path profile of each tag and requires densely deployed reference tags. In contrast, our scheme works on COTS devices and does not rely on any reference tags. Although both PinIt and our scheme leverage DTW metric and optimize its execution for tag localization, the targets of the DTW optimization in these two schemes are different. PinIt leverages derivative DTW (DDTW) technique to handle the power scaling problem, whereas our scheme optimizes the computational efficiency by applying the DTW on the coarse-grained representation of the phase profile.

8 Conclusions

In this paper, we propose the phase profiling approach to relative localization of RFID tags by exploiting the spatial-temporal dynamics in tag phase profiles. We show that relative localization can be achieved without the absolute location of tags. Our approach requires neither dedicated infrastructure nor special hardware. We implemented our approach and conducted experiments in two realistic case studies: locating misplaced books in a library and determining baggage ordering in an airport. The result shows that our approach can achieve high accuracy in realistic settings. This paper represents an early comprehensive study of relative localization of RFID tags. Our system can be used in a wide range of applications such as inventory control, asset management, and customer behavior tracking.

9 Acknowledge

We would like to thank the anonymous reviewers and our shepherd, Dr.Prabal Dutta, for providing valuable comments. This work is supported in part by the NSFC under grant numbers 61190110, 61171067, 61361166009, 61402338, 61472184, 61321491, 61272546, the NSFC Distinguished Young Scholars Program under Grant number 61125202, the Jiangsu Future Internet Program under Grant Number BY2013095-4-08, and the Jiangsu High-level Innovation and Entrepreneurship (Shuangchuang) Program.

References

[1] Impinj. `http://www.impinj.com`.

[2] The statistics of luggage missing in airport. `http://gadling.com/2010/03/26/airlines-losing-3000-bags-every-hour-of-every-day/`.

[3] Epc radio-frequency identity protocols. class-1 generation-2 uhf rfid. protocol for communications at 860 mhz to 960 mhz. *EPC global*, Jan 2005.

[4] Uhf class 1 gen 2 standard v. 1.0.9. *EPC global*, Jan 2005.

[5] S. Azzouzi, M. Cremer, U. Dettmar, R. Kronberger, and T. Knie. New measurement results for the localization of uhf rfid transponders using an angle of arrival (aoa) approach. In *Proceedings of RFID*, 2011.

[6] D. M. Dobkin. *The RF in RFID, Passive UHF RFID in Practice*. Elsevier, 2008.

[7] K. Donald, M. J. H. Jr, and P. Vaughan. Fast pattern matching in strings. *SIAM Journal on Computing*, 6, 1977.

[8] J. D. Griffin and G. D. Durgin. Complete link budgets for backscatter-radio and rfid systems. *IEEE Antennas and Propagation Magazine*, 2009.

[9] C. Hekimian-Williams, B. Grant, X. Liu, Z. Zhang, and P. Kumar. Accurate localization of rfid tags using phase difference. In *Proceedings of RFID*, 2010.

[10] C. Law, K. Lee, and K.-Y. Siu. Efficient memoryless protocol for tag identification (extended abstract). In *Proceedings of DIALM*, 2000.

[11] T. Liu, L. Yang, Q. Lin, Y. Guo, and Y. Liu. Anchor-free backscatter positioning for rfid tags with high accuracy. In *Proceedings of INFOCOM*, 2014.

[12] R. Miesen, F. Kirsch, and M. Vossiek. Holographic localization of passive uhf rfid transponders. In *Proceedings of RFID*, 2011.

[13] L. M. Ni, Y. Liu, Y. C. Lau, and A. P. Patil. Landmarc: Indoor location sensing using active rfid. *Wireless Networks*, 2004.

[14] P. Nikitin, R. Martinez, S. Ramamurthy, H. Leland, G. Spiess, and K. V. S. Rao. Phase based spatial identification of uhf rfid tags. In *Proceedings of RFID*, 2010.

[15] A. Parr, R. Miesen, and M. Vossiek. Inverse sar approach for localization of moving rfid tags. In *Procedings of RFID*, 2013.

[16] L. Shangguan, Z. Li, Z. Yang, M. Li, and Y. Liu. Otrack: Order tracking for luggage in mobile rfid systems. In *Proceedings of INFOCOM*, 2013.

[17] J. Wang, F. Adib, R. Knepper, D. Katabi, and D. Rus. Rf-compass: Robot object manipulation using rfids. In *Proceedings of MOBICOM*, 2013.

[18] J. Wang and D. Katabi. Dude, where's my card?: Rfid positioning that works with multipath and non-line of sight. In *Proceedings of SIGCOMM*, 2013.

[19] J. Wang, D. Vasisht, and D. Katabi. Rf-idraw: Virtual touch screen in the air using rf signals. In *Proceedings of SIGCOMM*, 2014.

[20] L. Yang, Y. Chen, X.-Y. Li, C. Xiao, M. Li, and Y. Liu. Tagoram: Real-time tracking of mobile rfid tags to high precision using cots devices. In *Proceedings of MOBICOM*, 2014.

[21] L. Yang, Y. Qi, J. Fang, X. Ding, T. Liu, and M. Li. Frogeye: Perception of the slightest tag motion. In *Proceedings of INFOCOM*, 2014.

[22] P. Zhang, J. Gummeson, and D. Ganesan. Blink: A high throughput link layer for backscatter communication. In *Proceedings of MOBISYS*, 2012.

[23] Y. Zhao, Y. Liu, and L. M. Ni. Vire: Active rfid-based localization using virtual reference elimination. In *Proceedings of ICPP*, 2007.

Ripple: Communicating through Physical Vibration

Nirupam Roy Mahanth Gowda Romit Roy Choudhury
University of Illinois at Urbana-Champaign

Abstract

This paper investigates the possibility of communicating through vibrations. By modulating the vibration motors available in all mobile phones, and decoding them through accelerometers, we aim to communicate small packets of information. Of course, this will not match the bit rates available through RF modalities, such as NFC or Bluetooth, which utilize a much larger bandwidth. However, where security is vital, vibratory communication may offer advantages. We develop *Ripple*, a system that achieves up to 200 bits/s of secure transmission using off-the-shelf vibration motor chips, and 80 bits/s on Android smartphones. This is an outcome of designing and integrating a range of techniques, including multi-carrier modulation, orthogonal vibration division, vibration braking, side-channel jamming, etc. Not all these techniques are novel; some are borrowed and suitably modified for our purposes, while others are unique to this relatively new platform of vibratory communication.

1 Introduction

Data communication has been studied over a wide range of modalities, including radio frequency (RF), acoustic, visible light, etc. This paper envisions vibration as a new mode of communication. We explore the possibility of using *vibration motors*, present in all cell phones today, as a transmitter, while *accelerometers*, also popular in mobile devices, as a receiver. By carefully regulating the vibrations at the transmitter, and sensing them through accelerometers, two mobile devices should be able to communicate via physical touch.

We are not the first to recognize this opportunity. Acoustic communication operates on the same fundamental principles and has been studied for decades (over air [24, 20] and under water [12]). In recent years, authors in [32] identified the possibility of using vibra-motors and accelerometers in mobile phones, as an opportunity to exchange information. The benefits were identified as security and zero-configuration, meaning that the two devices need not discover each other's addresses to communicate. The act of physical contact would serve as the implicit address. However, authors identified the drawbacks of such a system to be low bit rates ($\sim$ 5 bits/s), based on the "morse-code" style of ON/OFF communication with vibrations. Still, researchers conceived creative applications, including secure smartphone pairing and keyless access control [47].

This paper is aimed at improving the data rates of vibratory communication, as well as its security features. We design *Ripple*, a system that breaks away from the intuitive *morse-code* style ON/OFF pulses and engages techniques such as orthogonal multi-carrier modulation, gray coding, adaptive calibration, vibration braking, side-channel suppression, etc. While some techniques are borrowed from RF/acoustic communication, unique challenges (and opportunities) emerge from the vibra-motor/accelerometer platform, as well as from solid-materials on which they rest. For instance, the motor and the materials exhibit resonant frequencies that need to be adaptively suppressed; accelerometers sense vibration along 3 orthogonal axes, offering the opportunity to use them as parallel channels, with some degree of leakage. In addition to such techniques, we also design a receiver cradle – a wooden cantilever structure – that amplifies/dampens the vibrations in a desired way. A vibration based product in the future, say a point-of-sale equipment for credit card transactions, may potentially benefit from such a design.

From a security perspective, Ripple recognizes the threat of acoustic leakage due to vibration, i.e., an eavesdropper could listen to the sound of vibration and decode the transmitted bits. To thwart such side channel attacks, we design the transmitter to also listen to the sounds and adaptively play a synchronized acoustic signal (through its speaker) to cancel the sound. The transmitter also superimposes a jamming sequence, ultimately offering inherent protection from acoustic eavesdroppers. We observe that application layer securities may not apply in all such scenarios – public/symmetric key based encryption infrastructure may not scale to billions of phones and other use-cases such as internet of things (IoT). Blocking access to the signal, at the physical layer itself, is desirable in these spontaneous, peer-to-peer, and perhaps disconnected situations [41].

Its natural to wonder *what kind of applications will use vibratory communication, especially in light of NFC.* We do not have a killer app to propose, and even believe that most applications would prefer NFC, mainly due to its higher data rates (NFC uses 1.8MHz bandwidth achieving more than 100 Kbits/s, in contrast to 800Hz with today's vibra-motors). However, our hope is that bringing the vibratory bit rates to a respectable level – say credit card transactions in a second – may trigger new ideas and use-cases. In particular, strict security-sensitive applications may be the candidates. Despite the very short communication range in NFC, recent results [40, 28] confirm that security threats are real. Authors decode NFC transmissions from 1m away [14, 21, 22] and conjecture that high-gain beamforming antennas can further increase the separation. With the natural security benefits of touch-based communication (over RF), and supplemented with acoustic cancellation and jamming, we attempt to set a higher security bar for *Ripple*.

Moreover, the ubiquity of vibration motors in every cell phone, even in developing regions, presents an immediate market for vibratory communication. Peer to peer money exchange with recorded logs is a global problem, recently recognized by the Gates Foundation; hidden camera attacks on ATM kiosks have been rampant in many parts of India and south Asia [25]. Paying local cab drivers with phone-vibrations, or using phones as ATM cards can perhaps be of interest in developing countries. Clandestine operations may benefit where information need to be exchanged without leaving any trace in the wireless channel or in the Internet. Finally, if link capacity proves to be the only bottleneck, perhaps improved vibration motors can be included to mitigate it in the next phone models. While it's difficult to anticipate the needs of the future, we focus our attention on enabling and pushing forward this new modality of vibratory communication. To this end, our main contributions may be summarized as:

- Harnessing the vibration motor hardware and its functionalities, from a communication perspective.
- Developing an orthogonal multi-carrier communication stack using vibra-motor and accelerometer chips, and repeating the same for Samsung smartphones. Design decisions for the latter are different due to software/API limitations on smartphones, where vibra-motors were mainly integrated for simple alerts/notifications.
- Identifying acoustic side channel attacks and using signal cancellation and jamming to offer physical layer protection to eavesdropping.

2 Vibration Motors and Accelerometers

We begin with a high level overview of vibration motors and accelerometers (substantial details in [33, 34, 13]).

2.1 Vibration Motor

A vibration motor (also called "vibra-motor") is an electro-mechanical device that moves a metallic mass around a neutral position to generate vibrations. The motion is typically periodic and causes the center of mass (CoM) of the system to shift rhythmically. There are mainly two types of vibra-motors depending on their working principle:

(1) Eccentric Rotating Mass (ERM): This type of vibration generators uses a DC motor to rotate an eccentric mass around an axis as depicted in Figure 1(a). As the mass is not symmetric with respect to its axis of rotation, it causes the device to vibrate during the motion. Both the amplitude and frequency of vibration depend on the rotational speed of the motor, which can in turn be controlled through an input DC voltage. With increasing input voltages, both amplitude and frequency increase almost linearly and can be measured by an accelerometer.

(2) Linear Resonant Actuators (LRA) generate vibration by linear movement of a magnetic mass, as opposed to rotation in ERM (Figure 1)(b). With LRA, the mass is attached to a permanent magnet which is suspended near a coil, called "voice coil". Upon applying AC current to the motor, the coil also behaves like a magnet (due to the generated electromagnetic field) and causes the mass to be attracted or repelled, depending on the direction of the current. This generates vibration at the same frequency as the input AC signal, while the amplitude of vibration is determined by the signal's peak-to-peak voltage. Thus LRAs allow for regulating both the magnitude and frequency of vibration separately. Fortunately, most mobile phones today use LRA based vibra-motors.

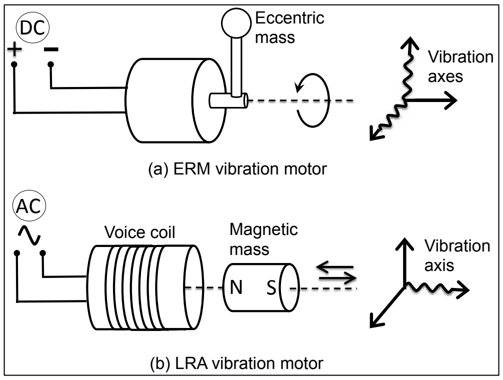


Figure 1: Basics of ERM and LRA vibra-motors.

Regulating Vibration

Ideally, a controller should be able to regulate the vibra-motor at fine granularities using any analog waveform. Unfortunately, micro-controllers produce digital voltage values limited to a few discrete levels. A popular technique to approximate analog signals with binary voltage levels is called Pulse Width Modulation (PWM) [8]. This technique is useful to drive analog devices with digital data without a digital to analog converter (DAC).

PWM based Motor Control: The core idea in PWM is to approximate any given voltage V by rapidly generating square pulses and configuring the pulse's duty cycle appropriately. For example, to create a $1V$ signal with binary voltage levels of $5V$ and $0V$, the duty cycle needs to be 20%. Now, if the period of the square pulse is made very small (i.e., high frequency), the effective output voltage will appear as $1V$. Towards this goal, the PWM frequency is typically set much higher than the response time of the target device so that the device experiences a continuous average voltage. Importantly, it is also possible to generate varying voltages with PWM, say a sine wave, by gradually changing the duty cycles in a sinusoidal fashion.

2.2 Accelerometer

The accelerometer is a micro electro-mechanical (MEMS) device that measures acceleration caused by motion. While the inner workings of accelerometers can vary [7], the core working principle pertains to a movable seismic mass that responds to the vibration of the object it is attached to. Capacitive accelerometers, shown in Figure 2, are perhaps most popular in smartphones today. When vibrated, the seismic mass moves between fixed electrodes, causing differences in the capacitance c_1 and c_2, ultimately producing a voltage proportional to the experienced vibration.

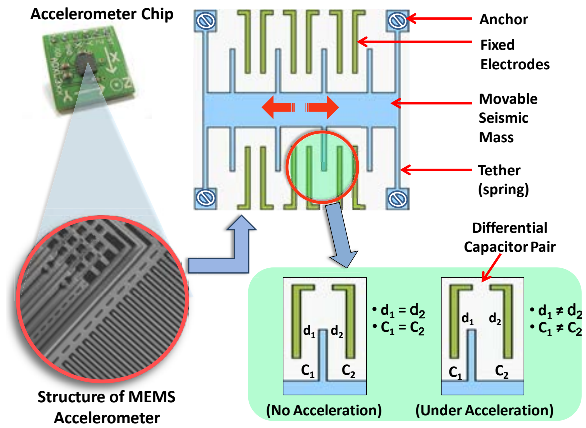


Figure 2: The internal architecture of MEMS accelerometer chip used in smartphones [19].

Sensing Acceleration

Modern accelerometers sense the movement of the seismic mass along 3 orthogonal axes, and report them as an $< X,Y,Z >$ tuple. The gravitational acceleration appears as a constant offset along the axis pointed towards the floor. The newest accelerometer chips support a wide range of adjustable sampling rates, typically from 100 mHz to $3.2KHz$. For this paper, we choose the ADXL345 [18] capacitive MEMS accelerometer, not only because it is used in most smartphones, but also because of programmability and frequency range.

3 Vibratory Transmission and Reception

Software/API limitations in smartphones prevent fully exploiting the vibra-motors and accelerometers. We design a custom hardware prototype using the same chips that smartphones use, and characterize/evaluate the system. We develop the constrained smartphone version in the next section.

3.1 Custom Hardware Setup

We control the vibra-motor and accelerometer through Arduino boards [1], an open source hardware development platform equipped with a ATmega328 8-bit RISC micro-controller [2]. Our first step is to precisely control the vibration frequency (and amplitude) through a time-varying sequence of voltage levels fed to the vibra-motor. Unfortunately, the micro-controller's output current fluctuates, leading to errors in the transmitted vibratory signals. Therefore, we power the vibra-motor with a stand-alone $6V$ DC power supply and use the Arduino micro-controller signal to operate a switch that regulates the voltage to the motor. We develop a simple circuit shown in Figure 3 – a NPN Darlington transistor (TIP122) serves as the switch and the controller signal goes to its base.

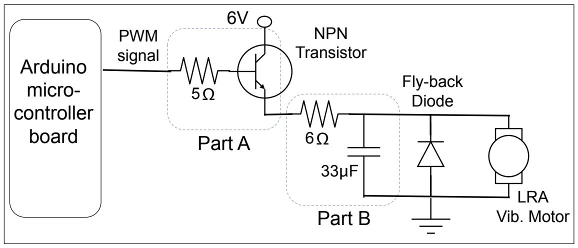


Figure 3: Transmitter hardware: the micro-controller controls a switch that regulates the $6V$ DC input.

Let's assume that we intend to regulate the vibra-motor in a sinusoidal fashion. We pre-load digital samples of the sine waveform into memory, and PWM uses them to determine the width of the square waves. When the sine wave frequency needs to be increased, the same digital samples need to be drawn at a faster rate and at precise timings. The switch uses the PWM output to regulate the $6V$ DC signal. We mitigate a number of engineering problems to run the set up correctly, including harmonic distortions due to the square pulses, spikes due to back EMF, etc. We move the PWM frequency to a high $32KHz$ and use an RC filter (part B Figure 3) to remove the distortions; we use a 1N4001 fly-back diode to smooth out the spikes. We omit further details in the interest of space.

The accelerometer receiver is also controlled through Arduino via the I2C protocol [44] at 115200 baud rate. We set the accelerometer's sampling rate to 1600Hz and 10 bit output resolution. While higher sampling rates are possible, we refrain from doing so since the micro-controller records the accelerometer data at a slower rate. In particular, the chip produces a sample per 0.625ms, but the micro-controller takes around $8 - 12$ms to periodically read and write in memory. We handle this with a FIFO mode of the accelerometer, such that the queued-up data is read in a burst. We also mount an on-board SD card to store data via the SPI protocol.

Figure 4 shows the accelerometer output when the vibra-motor is driven by the sinusoid input and made to touch the accelerometer. The final system functions correctly, and the platform is now ready for design and experimentation.

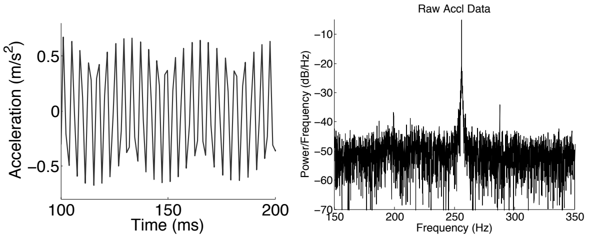


Figure 4: Accelerometer output (a) time and (b) frequency, when vibra-motor fed with a 250Hz sine wave.

3.2 Transmitter and Receiver Design

Ripple's design firmed up after multiple rounds of iterations. In the final version, the transmitter performs amplitude modulation on 10 different carrier signals uniformly spaced from 300 to $800Hz$ – each carrier is modulated with a bandwidth of $40Hz$. Further, the vibrations are also parallelized on orthogonal motion dimensions (X and Z) with appropriate signal cancellation. The design details are presented next.

Selecting the Carrier Signal

To reason about how data bits should be transmitted, we first carry out an analysis of the available spectrum. This available spectrum is actually bottlenecked by the maximum sampling rate of the accelerometer receiver – since this rate is 1600Hz, the highest frequency the transmitter can use is naturally 800Hz. Now, to test the system's frequency response in the $[0,800]$ band, we perform a "sine sweep" test. The transmitter, with the help of a waveform generator, produces continuously increasing frequencies from 1Hz to 800Hz with constant amplitude (the frequency increments are at 1Hz). Figure 5 shows the corresponding vibration magnitudes recorded by the accelerometer. Evidently, the response is weak up to 60Hz (called the "inert band"), followed by improvements till around 200Hz, followed by a large spike at around 231Hz. This spike is near the resonant frequency of the vibra-motor (confirmed in the data sheet).

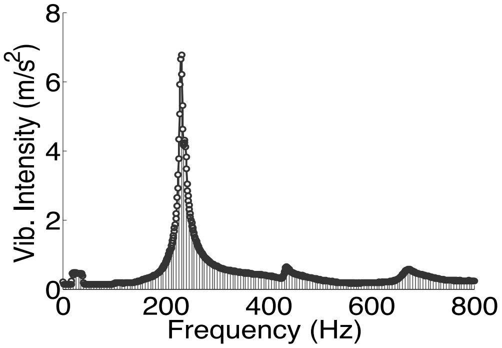


Figure 5: The vibra-motor's frequency response with the resonant frequency at around 231Hz.

Intuitively, frequencies near the resonant band can serve as good carriers for amplitude modulated data because of a larger vibration range. However, when we plot the frequency versus time spectrogram of the sine sweep test (Figure 6), we find that the vigorous vibration around the resonant frequency spills energy in almost the entire spectrum. Therefore, transmitting on the resonant band can be effective for a single carrier system, but the interference ruins the opportunity to transmit data in parallel carriers. In light of this, we define a "resonant band" of 100Hz around the peak, and move the carrier signals outside this band. We select 10 orthogonal carriers separated by 40Hz from the non-resonant frequencies between 300Hz and 800Hz. The 40Hz separation ensures the non-overlapping sidebands for the carriers, allowing reliable symbol recovery with software demodulation.

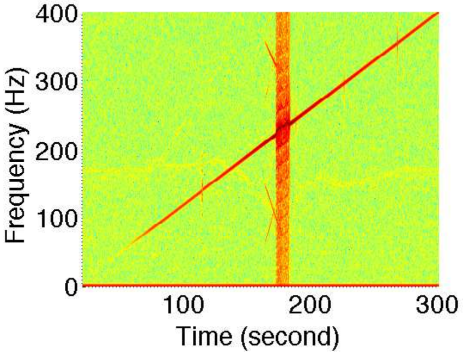


Figure 6: When excited with the resonant frequency, the vibra-motor spills energy across a wide frequency range.

Synchronization

Micro-controllers inject timing errors at various stages – variable delay in fetching digital samples from memory, during time-stamping the received samples, and due to oscillator/crystal frequency shifts with temperature. The timing errors manifest as fluctuations in vibration frequency, causing error in demodulation. To synchronize time between the transmitter and receiver, we introduce a pilot frequency at 70Hz and transmit it in parallel to data bits. We choose 70Hz to be above the inert band and lower than the resonant band. During reception, the receiver detects the pilot frequency, measures the offset in sampling rate, and interpolates the received signal by adjusting for this offset. Of course, this operation also corrects all other frequencies in the spectrum needed for demodulation.

(De)Modulating the Carrier Signal

The carrier frequencies are modulated with *Amplitude Shift Keying* (ASK) in light of its bandwidth efficiency and simplicity over Frequency Shift Keying (FSK). We modulate each of the 10 carriers with binary data at a symbol rate of 20Hz. To prevent inter-carrier interference, we shape the pulses with a raised cosine filter for each carrier individually; the modulated carriers are then combined and fed to the vibration motor transmitter. The receiver senses the energy in the pilot carrier, calibrates and synchronizes appropriately to identify the beginning of transmission. We again filter the received spectrum with (the same) raised cosine filter to isolate each carrier, and proceed to demodulate individual carriers separately. Figures 7(a) and (b) show a part of the spectrum before and after filtering, for an example carrier frequency at 405Hz. The demodulation is performed with envelope detection and precise sampling at bit intervals. We will evaluate this custom-designed system in Section 6 and show ∼200 bits/second data rates through vibration.

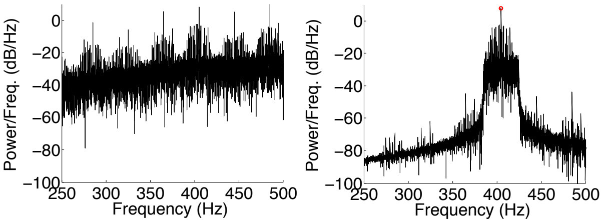


Figure 7: The spectrum (a) before and (b) after filtering for a single carrier frequency at 405Hz.

3.3 Orthogonal Vibration Dimensions

The above schemes, although adapted for vibra-motors, are grounded in the fundamentals of radio design. In an attempt to augment the bit rate, we observed that a unique property of accelerometers is its ability to detect vibration on 3 orthogonal dimensions (X, Y, and Z). Although vibra-motors only produce signals on a single dimension, perhaps multiple vibra-motors could be used in parallel. Unfortunately, due to some rigidity in our custom set up, accelerometer's motion along the X axis is minimal, precluding it for communication. Therefore, we orient two vibra-motors in the Y and Z dimensions and execute the exact multi-carrier amplitude modulated transmissions discussed above.

Measurements show that vibration from one dimension spills into the other. However, rather interestingly, this spilled interference exhibits a 180° phase lag with respect to the original signal, as well as an attenuation in the amplitude. Figure 8 shows an example in which the Z axis signal (solid black) has a spill on the Y axis, with a reversed phase and halved amplitude. The vice versa also occurs. Now, to remove Z's spilled interference and decode the Y signal, we scale the Y signal so that the interference matches Z's actual amplitude, and then add it to the Z signal. The Z signal is removed quite precisely, leaving an amplified version of Y, which is then decoded through the envelope detector. The reverse is performed with Z's signal, resulting in a 2x improvement in data rate, evaluated later.

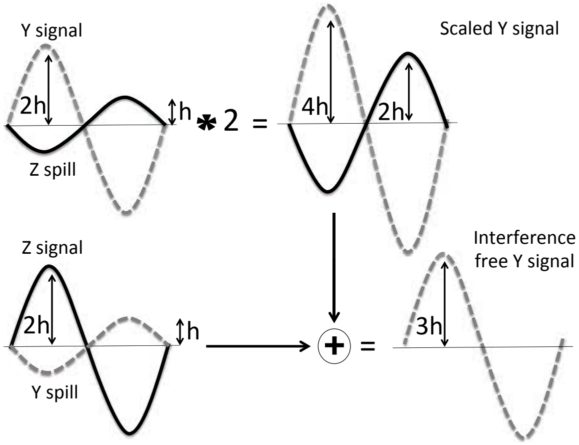


Figure 8: Orthogonal vibrations in X and Z axes.

4 Smartphone Prototype

This section shifts focus to vibratory communication on Android smartphones. Android is of interest since it offers APIs to a kernel level PWM driver for controlling the ON/OFF timings. We develop a user space module that leverages third-party kernel space APIs [5] to control the vibration amplitudes as well. However, this still does not match the custom set-up in the previous section. The PWM driver in Samsung smartphones is set to operate on the resonant band of the LRA vibra-motor, and the vibration frequency cannot be changed. This is understandable from the manufacturer's viewpoint, since vibra-motors are embedded to serve as a 1 bit alert to the user. However, for data communication, the nonlinear response at the resonant frequencies presents difficulties. Nonetheless, Ripple has to operate under these constraints and hence is limited to a single carrier frequency, modulated via amplitude modulation.

4.1 Smartphone Tx and Custom Rx

One advantage of the resonant frequency is that it offers a larger amplitude range, permitting *n*-ary symbols as opposed to binary (i.e., the amplitude range divided into *n* levels). To further amplify this range, we also design a custom smartphone cradle – a cantilever based wooden bridge-like framework – that in contact with the phone amplifies specific vibration frequencies. While we will evaluate performance without this cradle, we were curious if (deliberately designed) auxiliary objects bring benefits to vibratory communication. Figure 9 shows the design – when the transmitter phone is placed on a specific location on this bridge, and the accelerometer connected to the other end, we indeed observe improved SNR. The key idea here is to make the "channel" resonate along with the smartphone to improve transmission capacity. We elaborate on the cantilever based design next, followed by the communication techniques.

Cantilever based Receiver Setup

Observe that every object has a *natural frequency* [46] in which it vibrates. If an object is struck by a rod, say, it will vibrate at its natural frequency no matter how hard it is struck. The magnitude of the strike will increase the amplitude of vibration, but not its frequency. However, if a periodic force is applied at the same natural frequency of the object, the object exhibits amplified vibration – resonance. In our set-up, we use a 1 foot long wooden beam supported at one end, called a cantilever (Figure 9). The smartphone transmitter placed near the supported end, impinges a periodic force on the beam, calculated precisely based on the beam's resonant frequency (inversely proportional to $\sqrt{(}weight))$. We adjust the weight of the structure so that its natural frequency matches that of the phone's vibra-motor (which lies between 190Hz to 250Hz). This creates the desired resonance.

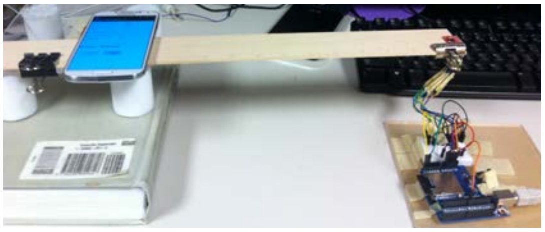


Figure 9: Cantilever based receiver platform for vibration amplification.

The accelerometer is attached at the unsupported end of the beam. Figure 10 plots the measured amplitude variation (over 3 axes of the accelerometer) as the smartphone is placed on different positions on the beam. We choose the position located 6 inches from the supported end, as it induces maximal amplification on all 3 axes of the accelerometer.

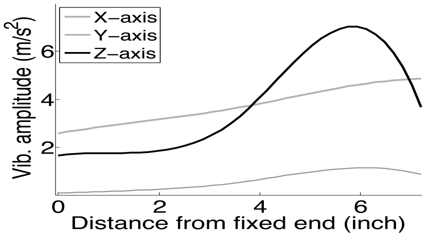


Figure 10: Vibration highest at a specific phone location.

Symbol Duration and the Ringing Effect

Ripple communicates through amplitude modulation – pulses of *n*-ary amplitudes (symbols) are modulated on the carrier frequency for a symbol duration. Ideally, the effect of a vibration should be completely limited within this symbol duration to avoid interference with the subsequent symbol (called inter-symbol interference). In practice, however, the vibration remains in the medium even after the driver stops the vibrator, known as the *ringing effect*. This is an outcome of inertia – the vibra-motor mass continues oscillating or rotating for some period after the driving voltage is turned off. Until this extended vibration dampens down substantially, the next symbol may get incorrectly demodulated (due to this heightened noise floor). Moreover, the free oscillation of the medium also contributes to ringing. Figure 11(a) shows a vibratory pulse of the smartphone, where the vibra-motor is activated from 20 to 50 ms. Importantly, the motor consumes 30 ms to overcome static inertia of the movable mass and reach its maximum vibration level. Once the voltage is turned off (at 50ms) the vibration dampens slowly and consumes another 70 ms to become negligible. This dictates the symbol duration to be around $30+70 = 100$ ms to avoid inter-symbol interference.

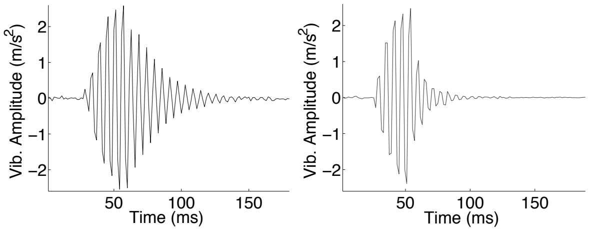


Figure 11: (a) Ringing effect in the channel. (b) Reduced ringing using a braking voltage.

Vibration Dampening

To push for greater capacity, we attempt to reduce the symbol duration by dampening the ringing vibration. The core observation is that the ringing duration is a function of the amplitude of the signal – a higher amplitude signal rings for a longer duration. If, however, the amplitude can be deliberately curbed, ringing will still occur but will decay faster. Based on this intuition, we apply a small *braking-voltage* to the vibra-motor right after the signal has been sampled by the demodulator

(30ms). This voltage is deliberately small so that it does not manifest into large vibrations, and is applied for 10ms. Once braking is turned off, we allow another 10ms for the tail of the ringing to die down, and then transmit the next symbol. Thus the symbol duration is 50ms now (half of the original) and there is still some vibration when we trigger the next symbol. While this adds slightly to the noise floor of the system, the benefits of a shorter symbol duration out-weighs the losses. Moreover, an advantage arises in energy consumption – triggering the vibra-motor from a cold start requires higher power. As we see later, activating it during the vibration tail saves energy.

(De)Modulation

The (de)modulation technique is mostly similar to a single carrier of the custom hardware prototype. The only difference is that it uses multiple levels of vibration amplitudes (up to 16), unlike the binary levels earlier. Figure 12 shows how we can vary the voltage levels (as a percentage of maximum input voltage) to achieve different vibration amplitudes. If adequately stable, the amplitude at each voltage level can serve as separate symbols. Given the linear amplitude slope from voltage levels 15 to 90%, we divide this range into n-ary equi-spaced amplitude levels, each corresponding to a symbol. However, due to various placements and/or orientations of the phone, this slope can vary to some degree. While this does not affect up to 8-ary communication, 16 symbols are susceptible to this because of inadequate gaps between adjacent amplitude levels. To cope, we use a preamble of two symbols. At the beginning of each packet the transmitter sends two symbols with the highest and lowest amplitudes (15 and 90). The receiver computes the slope from these two symbols, and calibrates all the other intermediate amplitude levels from them. The receiver then decodes the bits with a maximum likelihood based symbol detector.

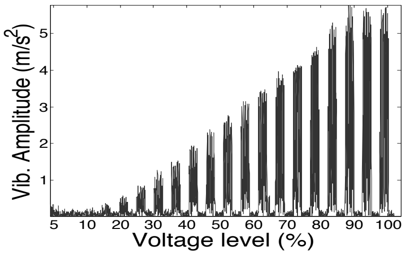


Figure 12: The change of vibration amplitude with the percentage of maximum input voltage.

5 Security

Vibrations produce sound and can leak information about the transmitted bits to an acoustic eavesdropper [29, 3, 9]. This section is aimed at designing techniques that thwart such side channel attacks. We design this as a real-time operation on the smartphone.

5.1 Acoustic Side Channel

The source of noise that actually leaks information is the rattling of the loosely-attached parts of the motor – the unbalanced mass and metals supporting it. Our experiments show that this *sound of vibration* (SoV) exhibits correlations of ∼0.7 with the modulated frequency of the data transmission. Although SoV decays quickly with distance, microphone arrays and other techniques can be employed to still extract information. Ripple attempts to prevent such attacks.

5.2 Canceling Sounds of Vibration (SoV)

One way to defend against eavesdropping is to jam the acoustic channel with a *pseudorandom noise* sequence, thus decreasing the SNR of the SoV. Since this jamming signal will not interfere with physical vibrations, it does not affect throughput. Upon implementation, we realized that the jamming signal was audible, and annoying to the ears. The more effective approach is perhaps to cancel/suppress the SoV from the source, and then jam faintly, to camouflage the residue.

Ideally, Ripple should produce an "anti-noise" signal that cancels out the SoV to ultimately create silence. The transmitter (and not the receiver) should generate this anti-noise since it knows the exact bit sequence that is the source of the SoV. Of course, acoustic noise cancellation is a well studied area – several headphones today use a microphone to capture ambient sounds and blends a negative version of it through the headphone speakers. The challenge of course is in detecting the ambient sound in real time and producing the precise negative (phase shifted) signals. However, unlike Ripple, headphones need to cancel the ambient noise only at the human ear, and not at all other locations around the human.

With Ripple, the problem is easier in the sense that the transmitter exactly knows the bit sequence that is causing the SoV. This can help in modeling the sound waveform ahead in time, and can potentially be synchronized. The issue, however, is that the SoV varies based on the material medium on which the phone is placed; also the SoV needs to be cancelled at all locations in the surrounding area. Further, the phase of the SoV remains unpredictable as it depends on the starting position of the mass in the vibra-motor and the delay to attain the full swing. Finally, Android offers little support for real-time audio processing [10], posing a challenge to develop SoV cancellation on off-the-shelf phones.

5.3 Ripple Cancel and Jam

The overall technique is composed of 3 sub-tasks: anti-noise modeling, phase alignment, and jamming.

(1) Anti-noise modeling
The core challenge is to model the analog SoV waveform corresponding to the data bits that will be transmitted through vibration. Since the motor's vibration amplitude and frequency are known (i.e., the carrier frequency), the first approximation of this model is simple to create. However, as mentioned earlier, the difficulty arises in not knowing how the unknown material (on which the phone is placed) will impact the SoV. Apart from the fundamental vibration frequency, the precise SoV signal depends also on the strength and count of the *overtones* produced by the material. To estimate this, the Ripple transmitter first transmits a short "preamble", listens to its FFT, and picks the *top-K* strongest overtones. These overtones are combined in the revised signal model. Finally, the actual data bits are modeled in the time domain, reversed in sign, and added to create the final "anti-noise" signal. This is ready to be played on the speaker, except that the phase of anti-noise needs to precisely match the SoV.

(2) Phase Alignment with Frequency Switch
Unfortunately, Android introduces a variable latency of up to 10ms to dispatch the audio data to the hardware. This is excessive since a 2.5ms lag can cause constructive interference between the anti-noise and the SoV. Fortunately, two observations help in this setting: (1) the audio continues playing at the specified sample rate without any significant fluctuation, and (2) the sample rate of the active audio stream can be changed in real-time. Thus, we can now control the frequency of the online audio by changing the playback sample rate.

We leverage this frequency control to match the phase of anti-noise with the SoV. The key idea is to start the anti-noise as close as possible to the SoV, but increase the sampling frequency such that the fundamental frequency of the anti-noise increase by δf. When this anti-noise combines in the air with the SoV, it creates the amplitude of the sound to vary because of the small difference in the fundamental frequencies. Obviously, the maximum suppression of the SoV occurs when the amplitude of this combined signal is at its minimum. The phase difference between the SoV and anti-noise is almost matched at this point. At exactly this "phase-lock" time, Ripple switches the fundamental frequency of the anti-noise to its original value (i.e., lower by δf). It recognizes this time instant by tracking the envelope of the combined signal and switching frequencies at the minimum point on the envelope. Figure 13 illustrates the various steps leading up to the frequency switch, and the sharp drop in signal amplitude. The suppressed signal remains at that level thereafter.

(3) Jamming
The cancellation is not perfect because the timing of the operations are not instantaneous; microphone and speaker noise also pollute the anti-noise waveforms, leaving a small residue. To prevent attacks on this residue, Ripple superimposes a jamming signal – the goal is to camouflage the sound residue. Conceptually it is simple, since a *pseudorandom noise* sequence can be added to the anti-noise waveform once it has phase-locked with the vibration sound. Unfortunately, Android does not allow loading a second signal on top of a signal that is already playing. Note that if we load the jamming signal upfront (along with the modeled anti-noise signal), the precise phase estimation will fail. We develop an engineering work-around. When modeling the anti-noise waveform, we also add the jamming noise sequence, but pre-pad the latter with a few *zeros*. Thus, when the SoV and anti-noise combine, the zeros still offer opportunities for detecting the time when the signals precisely cancel. We phase-lock at these times and the outcome is the residual signal from imperfect cancellation, plus the jamming sequence. We will show in the evaluation how the SoV's SNR degrades due to such cancellation and jamming, offering good protection to eavesdropping. Of course, the tradeoff is that we need a longer preamble now for this phase alignment process. However, this is only an issue arising from current Android APIs.

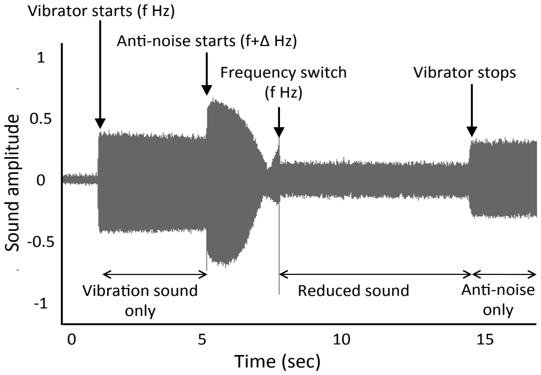


Figure 13: The anti-noise partially cancels the SoV, however, some mismatches result in some residual signal.

6 System Evaluation

We evaluate Ripple in three phases – the custom hardware, the smartphone prototype, and security.

6.1 Custom Hardware

Bit Error Rate (BER)

Recall that the custom hardware is composed of vibra-motors and accelerometer chips controlled by Arduino boards. We bring the two devices in contact and initiate packet transmission of various lengths (consuming between 1 to 10 seconds). Each packet contains pseudo-random binary bits at 20Hz symbol rate on 10 parallel carriers. The bits are demodulated at the receiver and compared against the ground truth. We repeat the exper-

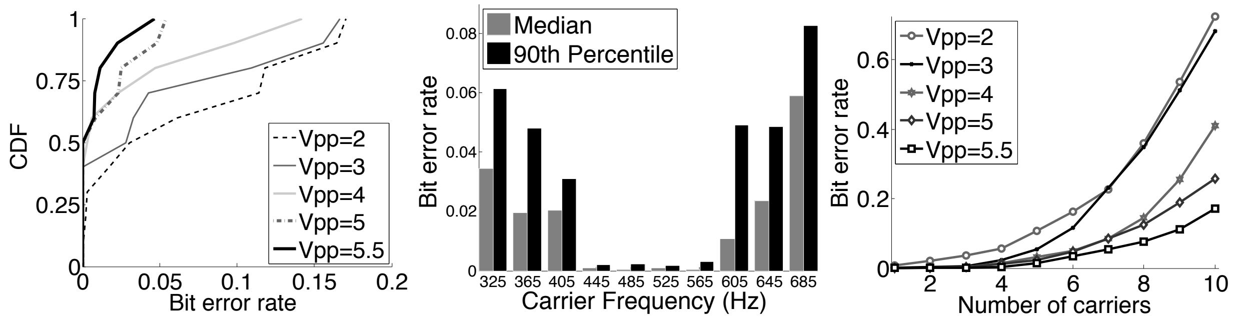


Figure 14: (a) BER as a function of the input signal peak-to-peak voltage (Vpp). Overall data rate ∼200 b/s. (b) Per-carrier BER across 10 frequencies. (c) BER as a function of no. of carriers used (each carrier bit rate = 20 bits/s).

iment for increasing signal energy (i.e., by varying the peak to peak signal voltage, Vpp, from 1V to 5V). Figure 14(a) plots the BER as a function of peak-to-peak input voltage (Vpp) to the vibra-motor and demonstrates how it diminishes with higher SNR. At the highest SNR, and aggregated over all carrier frequencies, Ripple achieves the 80^{th} percentile BER of 0.017 translating to an average bit rate of 196.6 bits/s.

Behavior of Carriers

In evaluating BERs across different carrier frequencies, we observe that not all carriers behave similarly. Figure 14(b) shows that carrier frequencies near the center of the spectrum perform consistently better than those near the edges. One of the reasons is *aliasing noise*. Ideally, the accelerometer should low-pass-filter the signal before sampling, to remove signal components higher than the Nyquist frequency. However, inexpensive accelerometers do not employ anti-aliasing filters, causing such undesirable effects. Carriers near the resonant band also experience higher noise due to the spilled-over energy.

Increasing the number of carriers will enable greater parallelism (bit rate), at the expense of higher BER per carrier. To characterize this tradeoff, we transmit data on increasing number of carriers, starting from the middle of our spectrum and activating carriers on both sides, one at a time. Figure 14(c) shows BER variations with increasing number of carriers, for varying signal energy (peak-to-peak voltage, Vpp). As each carrier operates at fixed 20Hz symbol rate, this also shows the bit rate vs BER characteristics of our system. Figure 15 zooms on the best four carriers.

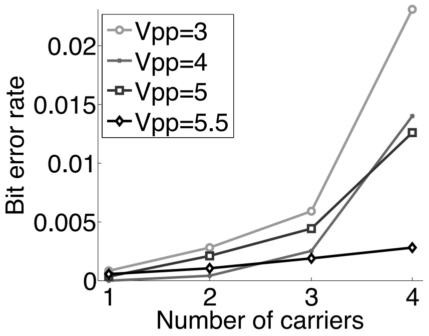


Figure 15: BER vs. number of carriers (4 carriers shown)

Temporal Stability

Given that vibra-motors and accelerometers are essentially mechanical systems, we intend to evaluate their properties when they are made to operate continuously for long durations. Given the low bit rates, this might be the case when relatively longer packets need to be transmitted. Towards this, we continuously transmit data for 50 sessions of 300 seconds each. Figure 16 plots the per-carrier BER (computed in the granularity of 10 second periods) of a randomly selected session – the Y axis shows each of the carriers and the X axis is time. The BERs vary between 0.02 near the center to 0.2 near the edge. Overall results, omitted for the interest of space, show no visible degradation in BER even after running for 300 seconds.

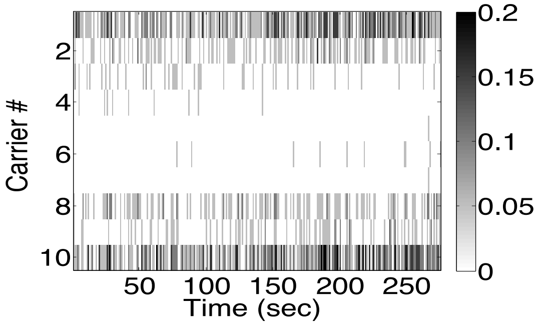


Figure 16: The BER per-carrier does not degrade after the motor is run for long durations.

Exploiting Vibration Dimensions

Recall that Ripple used 2 vibra-motors in parallel to exploit the orthogonality of vibrations along the Y and Z axes of the accelerometer. Figure 17(a) and (b) show the distribution of BER achieved across carrier frequencies on the Y and Z axes, respectively. We also attempt to push the limits by modulating greater than 20 bits/s, however, the BER begins to degrade. In light of this, Ripple achieves median capacity of around 400 bits/s (i.e., 20 bits/s per carrier x 10 carriers x 2 dimensions). While the tail of the BER distribution still needs improvement, we believe coding can be employed to mitigate some of it.

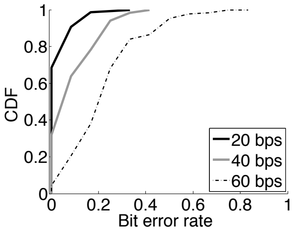


Figure 17: BER per carrier for parallel transmissions on orthogonal dimensions: (a) Y axis and (b) Z axis.

6.2 Smartphone Prototype

Calibration

Vibrations will vary across transactions due to phone orientation, humans holding it, different vibration medium, etc. As discussed earlier, the demodulator calibrates for these factors, but pays a penalty whenever the calibration is imperfect. We evaluate accuracy of calibration using the error between the estimated amplitude for a symbol, and the mean amplitude computed across all received symbols. Figure 18 plots the normalized error for various *n*-ary modulations – the normalization denominator is used as the difference between adjacent amplitudes.

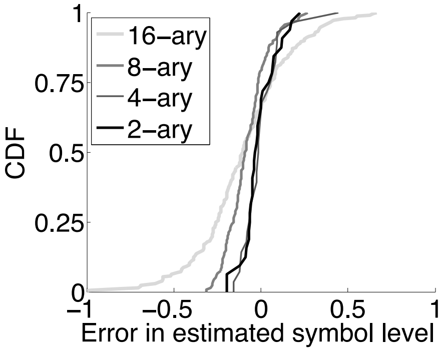


Figure 18: CDF of estimated symbol level error as a fraction of the mean inter-symbol difference.

BER with Smartphones

Figure 19(a) plots the confusion matrix of transmitted and received (or demodulated) symbols, for 16-ary modulation. While some errors occur, we observe that they are often the symbol adjacent to the one transmitted. In light of this, Ripple uses Gray codes to minimize such well-behaved errors. With these codes and calibration, Figure 19(b) shows the estimated BER for different bit rates, for each of the 4 modulation schemes. As comparison points, the "Basic" symbol detector uses predefined thresholds for each symbol and maps the received sample to the nearest amplitude. The "Ideal" scheme identifies the bits using the knowledge of all received symbols. Ripple's performs well even at higher bit rates, which is not the case with Basic.

Figure 19(c) shows the BER per symbol for 16-ary modulation, showing that symbols corresponding to the high vibration amplitudes experience higher errors. The reason is that the consistency of the vibration motor degrades at high amplitudes – we have verified this carefully by observing the distribution of received vibration amplitudes for large data traces.

Impact of Phone Orientation

The LRA vibra-motor inside Galaxy S4 generates linear vibration along one dimension – the teardown of the phone [11] shows the motor's axis aligned with the Z axis of the phone. Thus, an accelerometer should mostly witness vibration along the Z axis. The other two axes do not exhibit sufficient vibration at higher bit rates. This is verified in Table 1 where the first 4 data points are from when the phone is laid flat on top of the cantilever. However, once the phones are made to stand vertically or on the sides, its X and Y axes align with the accelerometers Z axis, causing an increase in errors. This suggests that the best contact points for the phones are their XY planes, mainly due to the orientation the vibration motor.

Table 1: BER with 16-ary for various orientations.

Orientation	Hor. A	Hor. B	Hor. C	Hor. D	Ver. A	Ver. B
Mean BER	0.025	0.029	0.002	0.029	0.197	0.178

Phone Held in Hand (No Cantilever)

We experiment a scenario in which the accelerometer based receiver is on the table, and the hand-held phone is made to touch the top of the receiver. The alignment is crudely along the Z axis. This setup adversely affects the system by (1) eliminating the amplitude gain due to the cantilever, and (2) the dampens vibration due to the hand's absorption. Figure 20 shows the results – unsurprisingly, the total vibration range is now smaller, pushing adjacent symbol levels to be closer to each other, resulting in higher BER.

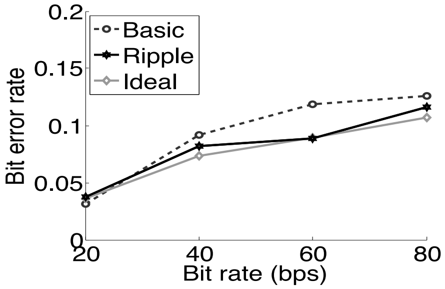


Figure 20: The BER with a hand-held phone.

6.3 Security

Acoustic Signal Leakage

To characterize the maximum acoustic leakage from vibrations, we run the vibra-motor at its highest intensity and record the SoV at various distances, using smartphone microphones sampled at 16KHz. This leakage is

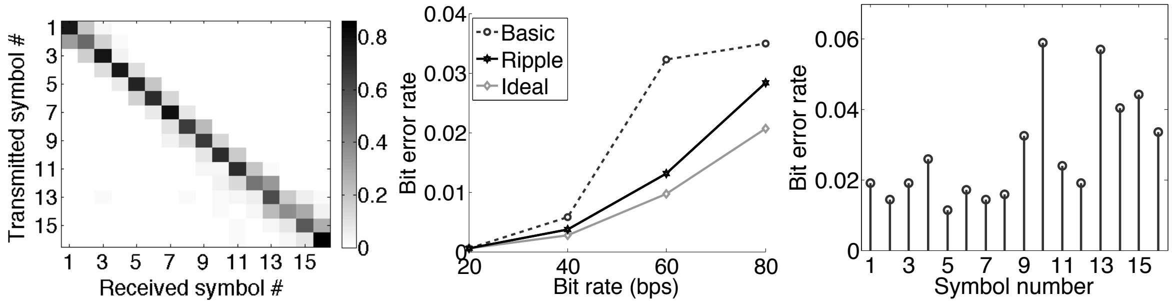


Figure 19: (a) This heat-map shows the confusion matrix of the transmitted and received symbols. (b) Ripple's BER compared to the Basic, Ideal. (c) Per symbol BER with 16-ary communication.

naturally far higher than a typical vibratory transmission (composed of various intensity levels), so mitigating the most severe leakage is stronger security. We also realize that the material on which the smartphone is placed matters, therefore, repeated the same experiment by placing the phone on (a) glass plate, (b) metal plate (aluminum), (c) on the top of another smartphone, and (d) our custom wooden cantilever setup. Figure 21 shows the contour plots for each scenario. Evidently, glass causes the strongest side channel leak, and wood is minimum. Following experiments are hence performed on glass.

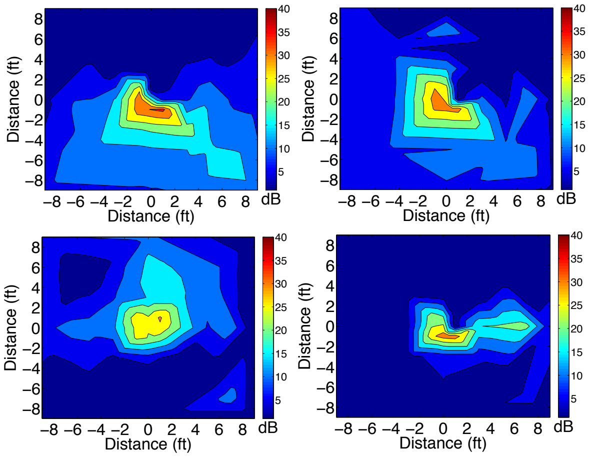


Figure 21: Acoustic side channel leakage on: (a) glass, (b) metal, (c) on another phone, and (d) wood.

Results indicate that the SoV is well below the socially acceptable noise level. At a distance of $2ft$, SoV is less than $25dB$, comparable to a soft whisper as per human perception of loudness [6]. We further quantify this by comparing SoV against the ambient noises recorded in 5 common locations – departmental store, inside a moving car, coffee shop, class room, and computer laboratory. Table 2 shows that the ratio remains close to 2.

Table 2: Ratio of power of SoV signals to ambient noise at public places.

Location	Dep. Store	Car	Coffee Shop	Class	Lab
Power ratio	1.57	1.81	2.01	2.10	2.31

Acoustic Leakage Cancellation

Recall that the Ripple receiver records the sound and produces a synchronized phase-shifted signal to cancel the sound, and superimposes a jamming sequence to further camouflage the leakage. Figure 22 shows the impact of cancellation using a ratio of the power of the residual signal to the original signal, measured at different distances. Evidently, the cancellation is better with increasing distance. This is because the generated "anti-noise" approximates the first few strong harmonics of the sound. However, the SoV also contains some other low-energy components that fade with distance making the anti-noise signal more similar to the vibration's sounds. Hence the cancellation is better at a distance, until around 4ft, after which residual signal drops below the noise floor and our calculated power becomes constant. The original signal also decreases but is still above the noise floor past 4ft, hence, the ratio increases.

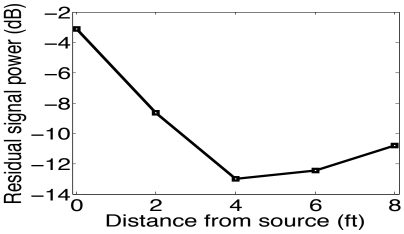


Figure 22: Ratio of residual to original signal power (in dB) at increasing distances from the source.

Acoustic Jamming

Ripple applies jamming to further camouflage any acoustic residue after the cancellation. To evaluate the lower bound of jamming efficiency, we make the experiment more favorable to the attacker. We transmit only two amplitude levels (binary data bits) at 10 bits per second. We place the phone on glass, the scenario that creates loudest sound. The eavesdropper microphone is placed as close as possible to the transmitter, without touching it. To quantify the efficacy of the jamming, we correlate the actual transmitted signal with the received jammed signal and plot the correlation coefficient in the Table 3. A high correlation coefficient indicates high probability of correctly decoding the message by the adversary, and the vice versa. The table shows the correlation values for various ratios of the jamming to signal power. Evident from the table, the correlation coefficient sharply decreases when Ripple increases the jamming power.

Table 3: The mean and std. dev. of the correlation coefficient for increasing jamming to signal power ratio.

Power ratio	0	0.4	0.8	1.2	1.6	2
Corr. mean	0.68	0.55	0.35	0.19	0.18	0.09
Corr. std. dev.	0.027	0.015	0.017	0.008	0.003	0.003

7 Limitations and Future Work

Needless to say, this paper is an early step – some aspects need deeper treatment, as discussed below.

Bounds and Optimality. We have not derived an upper bound on the capacity of vibratory communication, nor do we believe that our design decisions are optimal. We have taken an engineering approach and developed an end-to-end solution using techniques borrowed from RF/acoustic communication. Further work is needed to "tighten" the design towards optimality, including gains from coding and cancellation (on X, Y, Z dimensions).

Energy. Given that vibra-motors can be energy consuming, its important to characterize the energy versus throughput tradeoffs. For smartphone applications, vibrations are likely to be used occasionally for short exchanges, so perhaps energy is not a major hurdle. Nonetheless, when the phone battery is low, the ability to adapt can be a valuable feature.

Other Side-channels. An attacker could exploit the visual channel with a high-speed camera [17] to decode the vibratory bits. Even physical eavesdropping may be a threat, where the attacker sneakily attaches an accelerometer to the surface on which the Ripple devices are located. A probable solution to such attacks can be "vibratory jamming". Essentially, the receiver's vibra-motor could generate a pseudo-random jamming vibration while receiving the data from the transmitter. Of course, the transmitter is unaware of this and performs normal transmission. The net vibration video-recorded by the attacker's camera is actually the sum of two vibrations, hiding the actual transmitted bits. However, since the receiver knows the pseudo-random jamming sequence it has deliberately injected, it can cancel it out. Of course, this pseudo-random vibration should have enough power to create desirable entropy at the transmitter, else the eavesdropper can focus only on the transmitter's vibration. We leave the viability of these attacks and mitigations to future work.

8 Related Work

Vibration generation and sensing: Applications in haptic HCI for assisted learning, touch-augmented environments, and haptic learning have used vibrations for communication to humans [39, 23, 31, 42, 16]. However, the push for high communication data rates between vibrators and accelerometers is relatively unexplored. Off late, personal/environment sensing on mobile devices has gained research attention. Applications like (sp)iPhone [36] and TapPrints[38] demonstrate the ability to infer keystrokes through background motion sensing. While many more efforts are around activity recognition from vibration signatures, this paper aims to modulate vibration for communication.

Vibratory communication: The papers [45] and [32] are probably closest to Ripple. They both encode vibrations through ON-OFF keying, with ON/OFF durations in the range of a second (i.e., around 1 bits/s). This is adequate for applications like secure pairing between two smart phones, or sending a tiny URL over tens of seconds. However, unlike Ripple, they do not focus on the wide range of PHY and cross-layer radio design issues and possible security leaks. Dhwani [41] is an elegant work on acoustic NFC and addresses conceptually similar problems, however, their acoustic platform are appreciably different from Ripple.

Technologies like Bump [4, 37, 43, 15, 30, 27, 35] use accelerometer/vibrator-motor response to facilitate secure pairing between devices. However, these techniques are primarily designed to exchange small signatures, as opposed to the arbitrary data transmission in Ripple. As indicated by researchers [45, 26], the lack of the dynamic secret message in Bump-like techniques makes them less secure in the wild. These modes also require Internet connectivity and trusted third party servers to function, none of which is needed in Ripple.

9 Conclusion

This paper is an attempt to explore a new modality of communication – vibration. Through multi-carrier modulation, orthogonal vibration division, and leakage cancellation, our system, Ripple, is able to achieve 200 bits/s alongside a strong level of security against side channel attacks. While there is room for improvement, we believe this paper could serve as a stepping stone for exciting vibration-based technologies and applications.

Acknowledgement

We sincerely thank our shepherd Dr. Srinivasan Seshan and the anonymous reviewers for their valuable feedback. We are grateful to NSF for partially funding this research through grants CNS-1441638 and CNS-1430033.

References

[1] Arduino. http://arduino.cc.

[2] Atmel microcontroller. http://www.atmel.com/Images/doc8161.pdf.

[3] Audible noise reduction. http://www.danfoss.com/NR/rdonlyres/87FE3A20-0598-48C1-8A4B-75CFA8B25FC2/0/AudibleNoiseReduction.pdf.

[4] Bump technologies. http://bu.mp.

[5] Cyanogenmod. http://www.cyanogenmod.org.

[6] Loudness comparison. http://www.gcaudio.com/resources/howtos/loudness.html.

[7] Practical guide to accelerometers. http://www.sensr.com/pdf/practical-guide-to-accelerometers.pdf.

[8] Pulse width modulation. http://homepage.cem.itesm.mx/carbajal/Microcontrollers/ASSIGNMENTS/readings/ARTICLES/barr01_pwm.pdf.

[9] Reducing audible noise in vibration motor. http://www.precisionmicrodrives.com/tech-blog/2013/08/15/reducing-audible-noise-in-vibration-motors.

[10] Reducing audible noise in vibration motor. http://createdigitalmusic.com/2012/07/android-high-performance-audio-in-4-1-and-what-it-means-plus-libpd-goodness-today/.

[11] Samsung galaxy s4 teardown. https://www.ifixit.com/Teardown/Samsung+Galaxy+S4+Teardown/13947.

[12] BRADY, D., AND PREISIG, J. C. Underwater acoustic communications. *Wireless Communications: Signal Processing Perspectives 8* (1998), 330–379.

[13] BRAUER, J. R. *Magnetic actuators and sensors*. John Wiley & Sons, 2006.

[14] BROWN, T. W., DIAKOS, T., AND BRIFFA, J. A. Evaluating the eavesdropping range of varying magnetic field strengths in nfc standards. In *Antennas and Propagation (EuCAP), 2013 7th European Conference on* (2013), IEEE, pp. 3525–3528.

[15] CASTELLUCCIA, C., AND MUTAF, P. Shake them up!: a movement-based pairing protocol for cpu-constrained devices. In *Proceedings of the 3rd international conference on Mobile systems, applications, and services* (2005), ACM, pp. 51–64.

[16] CHO, Y.-J., YAND, T., AND KWON, D.-S. A new miniature smart actuator based on piezoelectric material and solenoid for mobile devices. In *The 5th International Conference on the Advanced Mechatronics, ICAM* (2010), pp. 615–620.

[17] DAVIS, A., RUBINSTEIN, M., WADHWA, N., MYSORE, G., DURAND, F., AND FREEMAN, W. T. The visual microphone: Passive recovery of sound from video. *ACM Transactions on Graphics (Proc. SIGGRAPH) 33*, 4 (2014), 79:1–79:10.

[18] DEVICES, A. Adxl345 datasheet. *USA: Analog Devices* (2010).

[19] DEY, S., ROY, N., XU, W., CHOUDHURY, R. R., AND NELAKUDITI, S. Accelprint: Imperfections of accelerometers make smartphones trackable. In *Proceedings of the Network and Distributed System Security Symposium (NDSS)* (2014).

[20] DHANANJAY, A., SHARMA, A., PAIK, M., CHEN, J., KUPPUSAMY, T. K., LI, J., AND SUBRAMANIAN, L. Hermes: data transmission over unknown voice channels. In *Proceedings of the sixteenth annual international conference on Mobile computing and networking* (2010), ACM, pp. 113–124.

[21] DIAKOS, T. P., BRIFFA, J. A., BROWN, T. W., AND WESEMEYER, S. Eavesdropping near-field contactless payments: a quantitative analysis. *The Journal of Engineering 1*, 1 (2013).

[22] EUN, H., LEE, H., AND OH, H. Conditional privacy preserving security protocol for nfc applications. *Consumer Electronics, IEEE Transactions on 59*, 1 (2013), 153–160.

[23] FEYGIN, D., KEEHNER, M., AND TENDICK, F. Haptic guidance: Experimental evaluation of a haptic training method for a perceptual motor skill. In *Haptic Interfaces for Virtual Environment and Teleoperator Systems, 2002. HAPTICS 2002. Proceedings. 10th Symposium on* (2002), IEEE, pp. 40–47.

[24] FUNKE, H. D. Acoustic body bus medical device communication system, May 19 1992. US Patent 5,113,859.

[25] GIRI, M., AND SINGH, D. Theoretical analysis of user authentication systems. *International Journal of Innovative Research and Development 2*, 12 (2013).

[26] HALEVI, T., AND SAXENA, N. On pairing constrained wireless devices based on secrecy of auxiliary channels: The case of acoustic eavesdropping. In *Proceedings of the 17th ACM conference on Computer and communications security* (2010), ACM, pp. 97–108.

[27] HALPERIN, D., HEYDT-BENJAMIN, T. S., RANSFORD, B., CLARK, S. S., DEFEND, B., MORGAN, W., FU, K., KOHNO, T., AND MAISEL, W. H. Pacemakers and implantable cardiac defibrillators: Software radio attacks and zero-power defenses. In *Security and Privacy, 2008. SP 2008. IEEE Symposium on* (2008), IEEE, pp. 129–142.

[28] HASELSTEINER, E., AND BREITFUSS, K. Security in near field communication (nfc). In *Workshop on RFID security* (2006), pp. 12–14.

[29] HENDERSHOT, J. Causes and sources of audible noise in electrical motors. In *Proc. 22nd Incremental Motion Control Systems and Devices Symposium* (1993), pp. 259–270.

[30] HOLMQUIST, L. E., MATTERN, F., SCHIELE, B., ALAHUHTA, P., BEIGL, M., AND GELLERSEN, H.-W. Smart-its friends: A technique for users to easily establish connections between smart artefacts. In *Ubicomp 2001: Ubiquitous Computing* (2001), Springer, pp. 116–122.

[31] HUANG, K., DO, E.-L., AND STARNER, T. Pianotouch: A wearable haptic piano instruction system for passive learning of piano skills. In *Wearable Computers, 2008. ISWC 2008. 12th IEEE International Symposium on* (2008), IEEE, pp. 41–44.

[32] HWANG, I., CHO, J., AND OH, S. Privacy-aware communication for smartphones using vibration. In *Embedded and Real-Time Computing Systems and Applications (RTCSA), 2012 IEEE 18th International Conference on* (2012), IEEE, pp. 447–452.

[33] KAAJAKARI, V. Practical mems: Design of microsystems, accelerometers, gyroscopes, rf mems, optical mems, and microfluidic systems. *Las Vegas, NV: Small Gear Publishing* (2009).

[34] KRISHNAN, R. *Electric motor drives: modeling, analysis, and control*. Prentice Hall, 2001.

[35] LESTER, J., HANNAFORD, B., AND BORRIELLO, G. are you with me?–using accelerometers to determine if two devices are carried by the same person. In *Pervasive computing*. Springer, 2004, pp. 33–50.

[36] MARQUARDT, P., VERMA, A., CARTER, H., AND TRAYNOR, P. (sp) iphone: decoding vibrations from nearby keyboards using mobile phone accelerometers. In *Proceedings of the 18th ACM conference on Computer and communications security* (2011), ACM, pp. 551–562.

[37] MAYRHOFER, R., AND GELLERSEN, H. Shake well before use: Intuitive and secure pairing of mobile devices. *Mobile Computing, IEEE Transactions on 8*, 6 (2009), 792–806.

[38] MILUZZO, E., VARSHAVSKY, A., BALAKRISHNAN, S., AND CHOUDHURY, R. R. Tapprints: your finger taps have fingerprints. In *Proceedings of the 10th international conference on Mobile systems, applications, and services* (2012), ACM, pp. 323–336.

[39] MORRIS, D., TAN, H. Z., BARBAGLI, F., CHANG, T., AND SALISBURY, K. Haptic feedback enhances force skill learning. In *WHC* (2007), vol. 7, pp. 21–26.

[40] MULLINER, C. Vulnerability analysis and attacks on nfc-enabled mobile phones. In *Availability, Reliability and Security, 2009. ARES'09. International Conference on* (2009), IEEE, pp. 695–700.

[41] NANDAKUMAR, R., CHINTALAPUDI, K. K., PADMANABHAN, V., AND VENKATESAN, R. Dhwani: secure peer-to-peer acoustic nfc. In *Proceedings of the ACM SIGCOMM 2013 conference on SIGCOMM* (2013), ACM, pp. 63–74.

[42] NIWA, M., YANAGIDA, Y., NOMA, H., HOSAKA, K., AND KUME, Y. Vibrotactile apparent movement by dc motors and voice-coil tactors. In *Proceedings of the 14th International Conference on Artificial Reality and Telexistence (ICAT)* (2004), pp. 126–131.

[43] SAXENA, N., AND WATT, J. H. Authentication technologies for the blind or visually impaired. In *Proceedings of the USENIX Workshop on Hot Topics in Security (HotSec)* (2009), vol. 9, p. 130.

[44] SEMICONDUCTORS, P. The i2c-bus specification. *Philips Semiconductors 9397*, 750 (2000), 00954.

[45] STUDER, A., PASSARO, T., AND BAUER, L. Don't bump, shake on it: The exploitation of a popular accelerometer-based smart phone exchange and its secure replacement. In *Proceedings of the 27th Annual Computer Security Applications Conference* (2011), ACM, pp. 333–342.

[46] WIDNALL, S. Lecture l19-vibration, normal modes, natural frequencies, instability. *Dynamics* (2009).

[47] YONEZAWA, T., NAKAHARA, H., AND TOKUDA, H. Vibconnect: A device collaboration interface using vibration. In *Embedded and Real-Time Computing Systems and Applications (RTCSA), 2011 IEEE 17th International Conference on* (2011), vol. 1, IEEE, pp. 121–125.

Multi-Person Localization via RF Body Reflections

Fadel Adib Zachary Kabelac Dina Katabi
Massachusetts Institute of Technology

Abstract– We have recently witnessed the emergence of RF-based indoor localization systems that can track user motion without requiring the user to hold or wear any device. These systems can localize a user and track his gestures by relying solely on the reflections of wireless signals off his body, and work even if the user is behind a wall or obstruction. However, in order for these systems to become practical, they need to address two main challenges: 1) They need to be able to operate in the presence of more than one user in the environment, and 2) they must be able to localize a user without requiring him to move or change his position.

This paper presents WiTrack2.0, a multi-person localization system that operates in multipath-rich indoor environments and pinpoints users' locations based purely on the reflections of wireless signals off their bodies. WiTrack2.0 can even localize static users, and does so by sensing the minute movements due to their breathing. We built a prototype of WiTrack2.0 and evaluated it in a standard office building. Our results show that it can localize up to five people simultaneously with a median accuracy of 11.7 cm in each of the x/y dimensions. Furthermore, WiTrack2.0 provides coarse tracking of body parts, identifying the direction of a pointing hand with a median error of 12.5°, for multiple users in the environment.

1 Introduction

Over the past decade, the networking community has made major advances in RF-based indoor localization [5, 34, 21, 26, 38, 17, 21, 11, 22], which led to systems that can localize a wireless device with centimeter-scale accuracy. Recently however the research community has realized that it is possible to localize a user, without requiring him to wear or carry a wireless device [25, 3]. Such a leap from device-based to device-free indoor localization can enable ubiquitous tracking of people and their gestures. For example, it can enable a smart home to continuously localize its occupants and adjust the heating in each room according to the number of people in it. It would also enable a smart home to track our hand gestures so that we may control appliances by pointing at them, or turn the TV with a wave of our arm. Device-free tracking can also be leveraged in many applications where it is either inconvenient or infeasible for the user to hold/wear a device such as in gaming and virtual reality, elderly monitoring, intrusion detection, and search and rescue missions [3].

Past work has taken initial steps towards this vision [3, 18, 7]. However, these proposals have fundamental limitations that render them impractical for natural home environments. Specifically, they either require covering the entire space with a dense, surveyed grid of sensors [18, 7] or they fail in the presence of multiple users in the environment [3]. Additionally, these past proposals are also limited in their ability to detect the presence of users. In particular, they either require the user to continuously move to detect his presence [3], or they need to perform extensive prior calibration or training [18, 7].

In this paper, we introduce WiTrack2.0, a device free localization system that transcends these limitations. Specifically, WiTrack2.0 accurately localizes multiple users in the environment. It does so by disentangling the reflections of wireless signals that bounce off their bodies. Furthermore, it neither requires prior calibration nor that the users move in order to localize them.

To achieve its goal, WiTrack2.0 has to deal with multiple challenges. As with traditional device-based localization, the most difficult challenge in indoor environments is the multipath effect [34, 17]. Specifically, wireless signals reflect off all objects in the environment making it hard to associate the incoming signal with a particular location. To overcome this challenge, past work [3] focuses on motion to capture signal reflections that change with time. It then assumes that only one person is present in the environment, and hence all motion can be attributed to him. However, if multiple people move in the environment or if the person is static, then this assumption no longer works.

To address this challenge, we observe that the indoor multipath varies significantly when it is measured from different vantage points. Hence, one can address this problem by positioning multiple transmit and receive antennas in the environment, and measuring the time of flight from each of these transmit-receive antenna pairs. However, the signals emitted from the different transmitters will reflect off the bodies of the all the users in the environment, and these reflections interfere with each other leading to wireless collisions. In §5, we show how WiTrack2.0 disentangles these interfering reflected signals to localize multiple users in the presence of heavy indoor multipath.

A second challenge that WiTrack2.0 has to address is related to the near-far problem. Specifically, reflections off the nearest person can have much more power than distant reflections, obfuscating the signal from distant people, and preventing their detection or tracking. To address this issue, we introduce Successive Silhouette Cancellation (SSC) an approach to address the near-far problem, which is inspired by successive interference cancellation. This technique starts by localizing the closest person, then eliminates his impact on the received signal, before proceeding to localize further users (who have weaker reflections). It repeats this process iteratively until it has localized all the users in a scene. Note, however, that each user is not a point reflector; hence, his wireless reflection has a complex structure that must be taken into account, as we describe in §6.

A third challenge that our system addresses is related to localizing static users. Specifically, past work that tracks human motion needs to eliminate reflections off static objects by subtracting consecutive measurements. However, this subtraction also results in eliminating the reflections off static users. To enable us to localize static users, we exploit the fact these users still move slightly due to their breathing. However, the breathing motion is fairly slow in comparison to body motion. Specifically, the chest moves by a sub-centimeter distance over a period of few seconds; in contrast, a human would pace indoors at 1 m/s. Hence, WiTrack2.0 processes the reflected signals at multiple time scales that enable it to accurately localize both types of movements as we describe in §7,

We have built a prototype of WiTrack2.0, using USRP software radios and an analog FMCW radio. We run experiments both in line-of-sight (LOS) scenarios and non-line-of-sight (NLOS) scenarios, where the device is in a different room and is tracking people's motion through the wall. Empirical results from over 300 experiments with 11 human subjects show the following:

- *Motion Tracking:* WiTrack2.0 accurately tracks the motion of up to four users simultaneously, without requiring the users to hold or wear any wireless device. In an area that spans 5 $m \times 7$ m, its median error across all users is 12.1cm in the x/y dimensions.
- *Localizing Static People:* By leveraging their breathing motion, WiTrack2.0 accurately localizes up to five static people in the environment. Its median error is 11.2 cm in the x/y dimensions across all the users in the scene.
- *Tracking Hand Movements:* WiTrack2.0's localization capability extends beyond tracking a user's body to tracking body parts. We leverage this capability to recognize concurrent gestures performed in 3D space by multiple users. In particular, we consider a gesture in which three users point in different directions at the same time. Our WiTrack2.0 prototype detects the pointing directions of all three users with a median accuracy of 10.3°.

Contributions: This paper demonstrates the first device-free RF-localization system that can accurately localize multiple people to centimeter-scale in indoor multipath-rich environments. This is enabled by a novel transmission protocol and signal processing algorithms which allow isolating and localizing different users in the environment. The paper also presents an evaluation of the system, showing that it can localize moving and static users in line-of-sight and through-wall settings with a median accuracy of 7-18 cm across all of these scenarios.

2 Related Work

Indoor Localization. WiTrack2.0 builds on a rich networking literature on indoor localization [5, 34, 21, 26, 38, 17, 21, 11, 22] which has focused on localizing wireless devices. In comparison to all of these works, however, WiTrack2.0 focuses on localizing users by relying purely on the reflections of RF signals off their bodies.

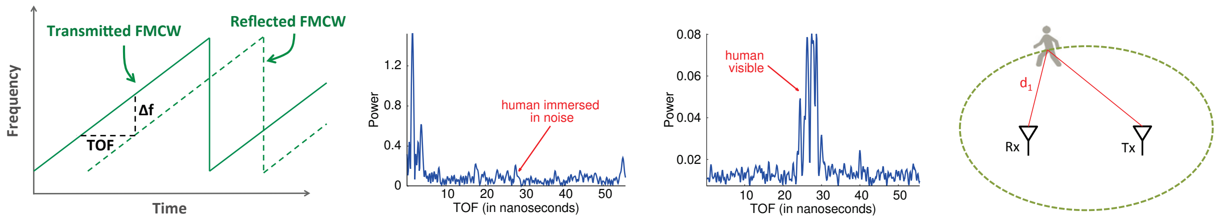
WiTrack2.0 is also related to proposals for device-free localization, which deploy a sensor network and measure the signal strength between different nodes to localize users [36, 40]. However, in comparison to these past proposals, WiTrack2.0 neither requires deploying a network of dozens to hundreds of sensors [36, 18] nor does it require extensive calibration [25, 24, 40]. Furthermore, because it relies on time-of-flight measurements, it can achieve a localization accuracy that is 10× higher than state-of-the-art RSSI-based systems [25, 24, 18, 7, 19].

WiTrack2.0's design builds on our prior work, WiTrack [3], which also used time of flight (TOF) measurements to achieve high localization accuracy, and which did not require prior environmental calibration. In comparison to WiTrack, which could localize only a single person and only if that person is moving, WiTrack2.0 can localize up to five users simultaneously, even if they are perfectly static (by relying on their breathing motion).

WiTrack2.0 is also related to non-localization systems that employ RF reflections off the human body [4, 20, 33, 35]. These systems can detect the presence of people or identify a handful of gestures or activities. However, unlike WiTrack2.0, they have no mechanism for obtaining the location of a person.

Radar Systems. WiTrack2.0 builds on past radar literature. In particular, it uses the FMCW (Frequency Modulated Carrier Waves) technique to obtain accurate time-of-flight measurements [15, 23, 31, 9]. However, its usage of FMCW has a key property that differentiates it from all prior designs: it transmits from multiple antennas *concurrently* while still allowing its receivers to isolate the reflections from each of the Tx antennas.

More importantly, none of the past work on radar addresses the issue of indoor multipath. Specifically, past work on see-through-wall radar has been tailored for usage in strictly military settings. Hence, it mostly operates in an open field with an erected wall [23, 10], or it focuses on detecting metallic objects which have significantly higher reflection coefficients than furniture, walls, or the human body [13, 28, 27]. Work tested on human subjects in indoor environments has focused on detecting the presence of humans rather than on accurately localizing them [16, 39]; in fact, these techniques acknowledge



(a) **FMCW provides TOF** (b) **TOF Profile** (c) **Background Subtraction** (d) **TOF to Ellipse**

Figure 1—Measuring Distances using FMCW. (a) shows the transmitted FMCW signal and its reflection. The TOF between the transmitted and received signals maps to a frequency shift Δf between them. (b) shows the TOF profile obtained after performing an FFT on the baseband FMCW signal. The profile plots the amount of reflected power at each TOF. (c) shows that a moving person's reflections pop up after background subtraction. (d) shows how a TOF measurement maps to a round-trip distance, which may correspond to any location on an ellipse whose foci are Tx and Rx.

that multipath leads to the well-known "ghosting effect", but ignore these effects since they do not prevent detecting the presence of a human. In comparison to the above work, WiTrack2.0 focuses on accurate localization of humans in daily indoor settings, and hence introduces two new techniques that enable it to address the heavy multipath in standard indoor environments: multi-shift FMCW, and successive silhouette cancellation.

Iterative Cancellation Frameworks. The framework of iteratively identifying and canceling out the strongest components of a signal is widely used in many domains. Naturally, however, the details of how the highest power component is identified and is eliminated varies from one application to another. In the communications community, we refer to such techniques as successive interference cancellation, and they have been used in a large number of applications such as ZigZag [12], VBLAST [37], and full duplex [6]. In the radio astronomy community, these techniques are referred to as CLEAN algorithms and, similarly, have a large number of instantiations [14, 29, 30]. Our work on successive silhouette cancellation also falls under this framework and is inspired by these algorithms. However, in comparison to all the past work, it focuses on identifying the reflections of the humans in the environment and canceling them by taking into account the different vantage points from which the time-of-flight is measured as well as the fact that the human body is not a point reflector.

3 Primer

This section provides necessary background regarding single-person motion tracking via RF body reflections.

The process of localizing a user based on radio reflections off her body has three steps: 1) obtaining time-of-flight (TOF) measurements to various reflectors in the environment; 2) eliminating TOF measurements due to reflections of static objects like walls and furniture; and 3) mapping the user's TOFs to a location.

Step 1: Obtaining TOF measurements to various reflectors in the environment. A typical way for measuring the time-of-flight (TOF) is to use a Frequency-Modulated Carrier Waves (FMCW) radio. An FMCW transmitter sends a narrowband signal (e.g., a few KHz) but makes the carrier frequency sweep linearly in time, as illustrated by the solid green line in Fig. 1(a). The reflected signal is a delayed version of the transmitted signal, which arrives after bouncing off a reflector, as shown by the dotted green line in Fig. 1(a). Because time and frequency are linearly related in FMCW, the delay between the two signals maps to a frequency shift Δf between them. Hence, the time-of-flight can be measured as the difference in frequency Δf divided by the slope of the sweep in Fig. 1(a):

$$TOF = \Delta f / slope \qquad (1)$$

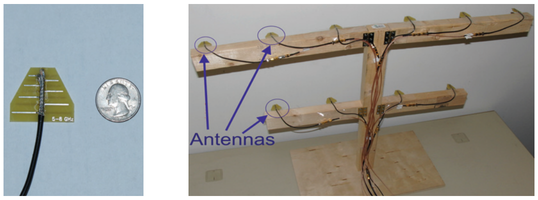
This description generalizes to an environment with multiple reflectors. Because wireless reflections add up linearly over the medium, the received signal is a linear combination of multiple reflections, each of them shifted by some Δf that corresponds to its TOF. Hence, one can extract all these TOFs by taking an FFT of the received signal. The output of the FFT gives us the ***TOF profile*** which we define as the reflected power we obtain at each possible TOF between the transmit antenna and receive antenna, as shown in Fig. 1(b).[1]

Step 2: Eliminating TOFs of static reflectors. To localize a human, we need to identify his/her reflections from those of other objects in the environment (e.g., walls and furniture). This may be done by leveraging the fact that the reflections of static objects remain constant over time. Hence, one can eliminate the power from static reflectors by performing background subtraction – i.e., by subtracting the output of the TOF profile in a given sweep from the TOF profile of the signal in the previous sweep. Fig. 1(b) and 1(c) show how background subtraction eliminates the power in static TOFs from the TOF profile, and allows one to notice the weak power resulting from a moving person.

Step 3: Mapping TOFs to distances. Recall that the TOF corresponds to the time it takes the signal to travel from the transmitter to a reflector and back to the receiver. Hence, we can compute the corresponding round-trip distance by multiplying this TOF by the speed of light C:

$$round\ trip\ distance = C \times TOF = C \times \frac{\Delta f}{slope} \qquad (2)$$

[1] The FFT is performed on the baseband FMCW signal – i.e., on the signal we obtain after mixing the received signal with the transmitted FMCW.



(a) **Antenna** (b) **Antenna Setup**

Figure 2—WiTrack2.0's Antennas and Setup. (a) shows one of WiTrack2.0's directional antenna ($3cm \times 3.4cm$) placed next to a quarter; (b) shows the antenna setup in our experiments, where antennas are mounted on a $2m \times 1m$ platform and arranged in a single vertical plane.

Knowing the round trip distance localizes the person to an ellipse whose foci are the transmit and receive antennas.

4 WITRACK2.0 OVERVIEW

WiTrack2.0 is a wireless system that can achieve highly accurate localization of multiple users in multipath-rich indoor environments, by relying purely on the reflections of wireless signals off the users' bodies. For static users, it localizes them based on their breathing, and can also localize the hand motions of multiple people, enabling a multi-user gesture-based interface.

WiTrack2.0 is a multi-antenna system consisting of five transmit antennas and five receive antennas, as shown in Fig. 2. The antennas are directional, stacked in a single plane, and mounted on a foldable platform as shown in Fig. 2(b). This arrangement is chosen because it enables see-through-wall applications, whereby all the antennas need to be lined up in a plane facing the wall of interest.

WiTrack2.0 operates by transmitting RF signals and capturing their reflections after they bounce off different users in the environment. Algorithmically, WiTrack2.0 has two main components: 1) Multi-shift FMCW, a technique that enables it to deal with multipath effects, and (2) Successive Silhouette Cancellation (SSC), an algorithm that allows WiTrack2.0 to overcome the near-far problem. The following sections describe these components.

5 MULTI-SHIFT FMCW

Multipath is the first challenge in accurate indoor localization. Specifically, not all reflections that survive background subtraction correspond to a moving person. This is because the signal reflected off the human body may also reflect off other objects in the environment before arriving at the receive antenna. As this person moves, this multipath reflection also moves with him and survives the background subtraction step. In single-user localization, one may eliminate this type of multipath by leveraging that these secondary reflections travel along a longer path before they arrive at the receive antenna. Specifically, by electing the smallest TOF after background subtraction, one may identify the round-trip distance to the user.

However, the above invariant does not hold in multi-person localization since different users are at different distances with respect to the antennas, and the multipath of a nearby user may arrive earlier than that of a more distant user, or even interfere with it. In this section, we explore this challenge in more details, and show how we can overcome it by obtaining time-of-flight measurements from different vantage points in the environment.

5.1 Addressing Multi-path in Multi-User Localization

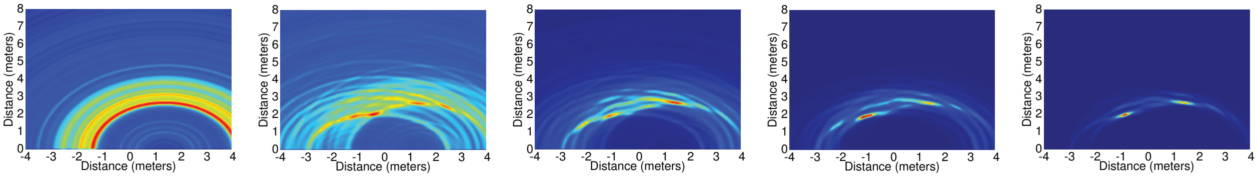
To explore the above challenge in practice, we run an experiment with two users in a 5×7 m furnished room (with tables, chairs, etc.) in a standard office building. We study what happens as we successively overlay ellipses from different transmit-receive pairs. Recall from §3 that each transmit-receive antenna pair provides us with a ***TOF profile*** – i.e., it tells us how much reflected power we obtain at each possible TOF between the transmit antenna and receive antenna (see Fig. 1(c)) – and that each such TOF corresponds to an ellipse in 2D (as in Fig. 1(d)).

Now let us map all TOFs in a TOF profile to the corresponding round trip distances using Eq. 2, and hence the resulting ellipses. This process produces a heatmap like the one in Fig. 3(a), where the x and y axes correspond to the plane of motion. For each ellipse in the heatmap, the color in the image reflects the amount of received power at the corresponding TOF. Hence, the ellipse in red corresponds to a strong reflector in the environment. The orange, yellow, and green ellipses correspond to weaker reflections respectively; these reflections could either be due to another person, multi-path reflections of the first person, or noise. The blue regions in the background correspond to the absence of reflections from those areas.

Note that the heatmap shows a pattern of half-ellipses; the foci of these ellipses are the transmit and receive antennas, both of which are placed along the $y = 0$ axis. The reason we only show the upper half of the ellipses is that we are using directional antennas and we focus them towards the positive y direction. Hence, we know that we do not receive reflections from behind the antennas.

Fig. 3(a) shows the ellipses corresponding to the TOF profiles from one Tx-Rx pair. Now, let us see what happens when we superimpose the heatmaps obtained from two Tx-Rx pairs. Fig. 3(b) shows the heatmap we obtain when we overlay the ellipses of the first transmit-receive pair with those from a second pair. We can now see two patterns of ellipses in the figure, the first pattern resulting from the TOFs of the first pair, and the second pattern due to the TOFs of the second pair. These ellipses intersect in multiple locations, resulting in red or orange regions, which suggest a higher probability for a reflector to be in those regions. Recall that there are two people in this experiment. However, Fig. 3(b) is not enough to identify the locations of these two people.

Figs. 3(c) and 3(d) show the result of overlaying ellipses from three and four Tx-Rx pairs respectively. The figures show how the noise and multi-path from different



(a) **One Tx-Rx pair** (b) **Two Tx-Rx pairs** (c) **Three Tx-Rx pairs** (d) **Four Tx-Rx pairs** (e) **Five Tx-Rx pairs**

Figure 3—Increasing the Number of Tx-Rx pairs enables Localizing Multiple Users. The figure shows the heatmaps obtained from combining TOF profiles of multiple Tx-Rx antenna pairs in the presence of two users. The x/y axes of each heatmap correspond to the real world x/y dimensions.

antennas averages out to result in a dark blue background. This is because different Tx-Rx pairs have different perspectives of the indoor environment; hence, they do not observe the same noise or multi-path reflections. As a result, the more we overlay heatmaps from different Tx-Rx pairs, the dimmer the multipath effect, and the clearer the candidate locations for the two people in the environment.

Next, we overlay the ellipses from five transmit-receive pairs and show the resulting heatmap in Fig. 3(e). We can now clearly see two bright spots in the heatmap: one is red and the other is orange, whereas the rest of the heatmap is mostly a navy blue background indicating the absence of reflectors. Hence, in this experiment, we are able to localize the two users using TOF measurements from five Tx-Rx pairs. Combining these measurements together allowed us to eliminate the multipath effects and localize the two people passively using their reflections.

Summary: As the number of users increases, we need TOF measurements from a larger number of Tx-Rx pairs to localize them, and extract their reflections from multipath. For the case of two users, we have seen a scenario whereby the TOFs of five Tx-Rx pairs were sufficient to accurately localize both of them. In general, the exact number would depend on multipath and noise in the environment as well as on the number of users we wish to localize. These observations motivate a mechanism that can provide us with a large number of Tx-Rx pairs while scaling with the number of users in the environment.

5.2 The Design of Multi-shift FMCW

In the previous section, we showed that we can localize two people by overlaying many heatmaps obtained from mapping the TOF profiles of multiple Tx-Rx pairs to the corresponding ellipses. But how do we obtain TOFs from many Tx-Rx pairs? One option is to use one FMCW transmitter and a large number of receivers. In this case, to obtain N Tx-Rx pairs, we would need one transmitter and N receivers. The problem with this approach is that it needs a large number of receivers, and hence does not scale well as we add more users to the environment.

A more appealing option is to use multiple FMCW transmit and receive antennas. Since the signal transmitted from each transmit antenna is received by all receive antennas, this allows us to obtain N Tx-Rx pairs using only $\sqrt{N}$ transmit antennas and $\sqrt{N}$ receive antennas.

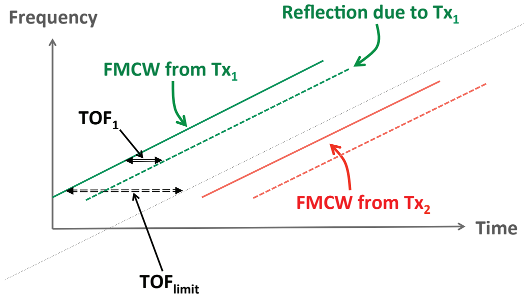


Figure 4—Multi-shift FMCW. WiTrack2.0 transmits FMCW signals from different transmit antennas after inserting virtual delays between them. Each delay must be larger than the highest time-of-flight (TOF_{limit}) due to objects in the environment.

However, the problem with this approach is that the signals from the different FMCW transmitters will interfere with each other over the wireless medium, and this interference will lead to localization errors. To see why this is true, consider a simple example where we want to localize a user, and we have two transmit antennas, Tx1 and Tx2, and one receive antenna Rx. The receive antenna will receive two reflections – one due to the signal transmitted from Tx1, and another due to Tx2's signal. Hence, its TOF profile will contain two spikes referring to two time-of-flight measurements TOF_1 and TOF_2.

With two TOFs, we should be able to localize a single user based on the intersection of the resulting ellipses. However, the receiver has no idea which TOF corresponds to the reflection of the FMWC signal generated from Tx1 and which corresponds to the reflection of the FMCW signal generated by Tx2. Not knowing the correct Tx means that we do not know the foci of the two ellipses and hence cannot localize. For example, if we incorrectly associate TOF_1 with Tx2 and TOF_2 with Tx1, we will generate a wrong set of ellipses, and localize the person to an incorrect location. Further, this problem becomes more complicated as we add more transmit antennas to the system. Therefore, to localize the user, WiTrack2.0 needs a mechanism to associate these TOF measurements with their corresponding transmit antennas.

We address this challenge by leveraging the structure of the FMCW signal. Recall that FMCW consists of a continuous linear frequency sweep as shown by the green line in Fig. 4. When the FMCW signal hits a body, it reflects back with a delay that corresponds to the body's TOF. Now, let us say TOF_{limit} is the maximum TOF

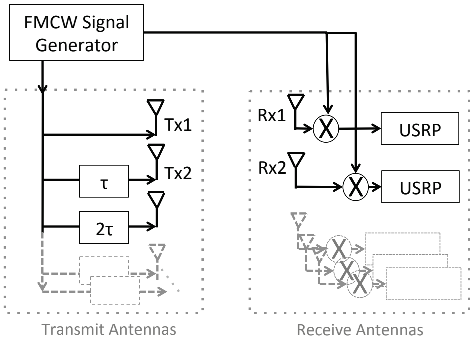


Figure 5—Multi-shift FMCW Architecture. The generated FMCW signal is fed to multiple transmit antennas via different delay lines. At the receive side, the TOF measurements from the different antennas are combined to obtain the 2D heatmaps.

that we expect in the typical indoor environment where WiTrack2.0 operates. We can delay the FMCW signal from the second transmitter by $\tau > TOF_{limit}$ so that all TOFs from the second transmitter are shifted by τ with respect to those from the first transmitter, as shown by the red line in Fig. 4. Thus, we can prevent the various FMCW signals from interfering by ensuring that each transmitted FMCW signal is time shifted with respect to the others, and those shifts are significantly larger than the time-of-flight to objects in the environment. We refer to this design as Multi-shift FMCW.

As a result, the receiver would still compute two TOF measurements: the first measurement (from Tx1) would be TOF_1, and the second measurement (from Tx2) would be $TOF_2' = TOF_2 + \tau$. Knowing that the TOF measurements from Tx2 will always be larger than τ, WiTrack2.0 determines that TOF_1 is due to the signal transmitted by Tx1, and TOF_2' is due to the signal transmitted by Tx2.

This idea can be extended to more than two Tx antennas, as shown in Fig. 5. Specifically, we can transmit the FMCW signal directly over the air from Tx1, then shift it by τ and transmit it from Tx2, then shift it by 2τ and transmit it from Tx3, and so on. At the receive side, all TOFs between 0 and τ are mapped to Tx1, whereas distances between τ and 2τ are mapped to Tx2, and so on.

Summary: Multi-shift FMCW has two components: the first component allows us to obtain TOF measurements from a large number of Tx-Rx pairs; the second component operates on the TOFs obtained from these different Tx-Rx pairs by superimposing them into a 2D heatmap, which allows us to localize multiple users in the scene.

6 Successive Silhouette Cancellation

With multi-shift FMCW, we obtain TOF profiles from a large number of Tx-Rx pairs, map them to 2D heatmaps, overlay the heatmaps, and start identifying users' locations. However, in practice this is not sufficient because different users will exhibit the near-far problem. Specifically, reflections of a nearby user are much stronger than reflections of a faraway user or one behind an obstruction.

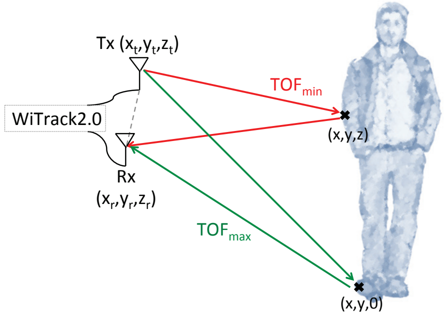


Figure 7—Finding TOF_{min} and TOF_{max}. TOF_{min} is determined by the round-trip distance from the Tx-Rx pair to the closest point on the person's body. Since the antennas are elevated, TOF_{max} is typically due to the round-trip distance to the person's feet.

Fig. 6(a) illustrates this challenge. It shows the 2D heatmap obtained in the presence of four persons in the environment. The heatmap allows us to localize only two of these persons: one is clearly visible at $(0.5, 2)$, and another is fairly visible at $(-0.5, 1.3)$. The other two people, who are farther away from WiTrack2.0, are completely overwhelmed by the power of the first two persons.

To deal with this near-far problem, rather than localizing all users in one shot, WiTrack2.0 performs Successive Silhouette Cancellation (SSC) which consists of 4 steps:

1. *SSC Detection:* finds the location of the strongest user by overlaying the heatmaps of all Tx-Rx pairs.
2. *SSC Re-mapping:* maps a person's location to the set of TOFs that would have generated that location at each transmit-receive pair.
3. *SSC Cancellation:* cancels the impact of the person on the TOF profiles of all Tx-Rx pairs.
4. *Iteration:* re-computes the heatmaps using the TOF profiles after cancellation, overlays them, and proceeds to find the next strongest reflector.

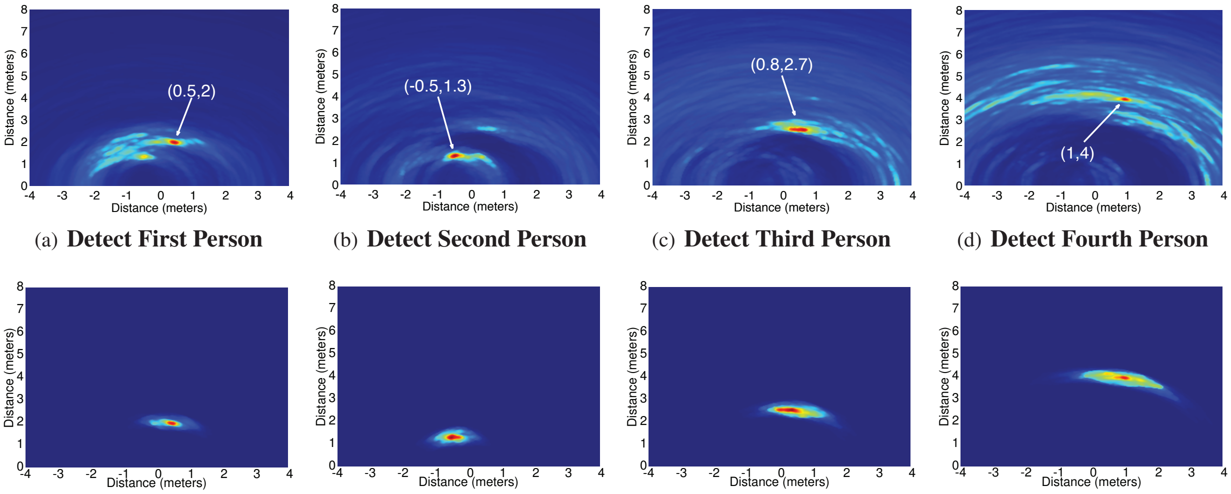
We now describe each of these steps in detail by walking through the example with four persons shown in Fig. 6.

SSC Detection. In the first step, SSC finds the location of the highest power reflector in the 2D heatmap of Fig. 6(a). In this example, the highest power is at $(0.5, 2)$, indicating that there is a person in that location.

SSC Re-mapping. Given the (x, y) coordinates of the person, we map his location back to the corresponding TOF at each transmit-receive pair. Keep in mind that each person is not a point reflector; hence, we need to estimate the spread of reflections off his entire body on the TOF profile of each transmit-receive pair.

To see how we can do this, let us look at the illustration in Fig. 7 to understand the effect of a person's body on one transmit-receive pair. The signal transmitted from the transmit antenna will reflect off different points on the person's body before arriving at the receive antenna. Thus, the person's reflections will appear between some TOF_{min} and TOF_{max} in the TOF profile at the Rx antenna.

Note that TOF_{min} and TOF_{max} are bounded by the closest and furthest points respectively on a person's body



(a) **Detect First Person** (b) **Detect Second Person** (c) **Detect Third Person** (d) **Detect Fourth Person**

(e) **Focus on First Person** (f) **Focus on Second Person** (g) **Focus on Third Person** (h) **Focus on Fourth Person**

Figure 6—Successive Silhouette Cancellation. (a) shows the 2D heatmap obtained by combining all the TOFs in the presence of four users. (b)-(d) show the heatmaps obtained after canceling out the first, second, and third user respectively. (e)-(h) show the result of the SSC focusing step on each of the users, and how it enables us to accurately localize each person while eliminating interference from all other users.

from the transmit-receive antenna pair. Let us first focus on how we can obtain TOF_{min}. By definition, the closest point on the person's body is the one that corresponds to the shortest round-trip distance to the Tx-Rx pair, where the round-trip distance is the summation of the forward path from Tx to that point and the path from that point back to Rx. Formally, for a Tx antenna at $(x_t, 0, z_t)$, an Rx antenna at $(x_r, 0, z_r)$,[2] we can compute d_{min} as:

$$\min_z \sqrt{(x_t - x)^2 + y^2 + (z_t - z)^2} + \sqrt{(x_r - x)^2 + y^2 + (z_r - z)^2} \tag{3}$$

where (x, y, z) is any reflection point on the user's body. One can show that this expression is minimized when:

$$\frac{z - z_t}{z - z_r} = -\sqrt{\frac{(x_t - x)^2 + y^2}{(x_r - x)^2 + y^2}} \tag{4}$$

Hence, using the detected (x, y) position, we can solve for z then substitute in Eq. 3 to obtain d_{min}.

Similarly, TOF_{max} is bounded by the round-trip distance to point on the person's body that is furthest from the Tx-Rx pair. Again, the x and y coordinates of the furthest point are determined by the person's location from the SSC Detection step. However, we still need to figure out the z coordinate of this point. Since the transmitter and receiver are both raised above the ground (at around 1.2 meters above the ground), the furthest point from the Tx-Rx pair is typically at the person's feet. Therefore, we know that the coordinates of this point are $(x, y, 0)$, and hence we can compute d_{max} as:

$$d_{max} = \sqrt{(x_t - x)^2 + (y)^2 + z_t^2} + \sqrt{(x_r - x)^2 + (y)^2 + z_r^2}.$$

Finally, we can map d_{min} and d_{max} to TOF_{min} and TOF_{max} by dividing them by the speed of light C.

SSC Cancellation. The next step is to use TOF_{min} and TOF_{max} to cancel the person's reflections from the TOF profiles of each transmit-receive pair. To do that, we take a conservative approach and remove the power in all TOFs between TOF_{min} and TOF_{max} within that profile. Of course, this means that we might also be partially canceling out the reflections of another person who happens to have a similar time of flight to this Tx-Rx pair. However, we rely on the fact that multi-shift FMCW provides a large number of TOF profiles from many Tx-Rx pairs. Hence, even if we cancel out the power in the TOF of a person with respect to a particular Tx-Rx pair, each person will continue to have a sufficient number of TOF measurements from the rest of the antennas.

We repeat the process of computing TOF_{min} and TOF_{max} with respect of each Tx-Rx pair and cancelling the power in that range, until we have eliminated any power from the recently localized person.

Iteration. We proceed to localize the next person. This is done by regenerating the heatmaps from the updated TOF profiles and overlaying them. Fig. 6(b) shows the obtained image after performing this procedure for the first person. Now, a person at $(-0.5, 1.3)$ becomes the strongest reflector in the scene.

We repeat the same procedure for this user, canceling out his interference, then reconstructing a 2D heatmap in Fig. 6(c) using the remaining TOF measurements. Now, the person with the strongest reflection is at $(0.8, 2.7)$. Note that this heatmap is noisier than Figs. 6(a) and 6(b) because now we are dealing with a more distant person.

WiTrack2.0 repeats the same cancellation procedure for the third person and constructs the 2D heatmap in Fig. 6(d). The figure shows a strong reflection at $(1, 4)$. Recall that our antennas are placed along the $y = 0$ axis,

[2] Recall that all the antennas are in the vertical plane $y = 0$, which is parallel to a person's standing height.

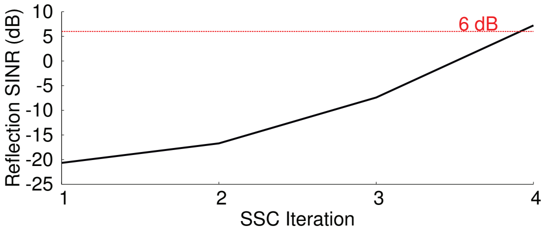


Figure 8—SINR of the Farthest User Throughout SSC Iterations. The figure shows how the SINR of the farthest user increases with each iteration of the SSC algorithm. After the first, second and third person are removed from the heatmap images in Figs.6(a)-(c), the SINR of the fourth person increases to 7dB, allowing us to detect his presence.

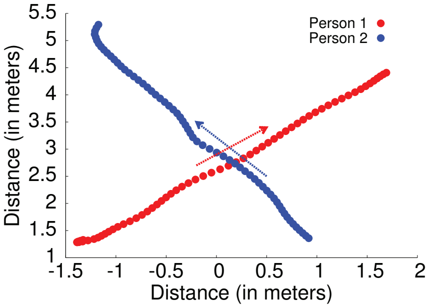


Figure 9—Disentangling Crossing Paths. When two people cross paths, they typically keep going along the same direction they were going before their paths crossed.

which means that this is indeed the furthest person in the scene. Also note that the heatmap is now even noisier. This is expected because the furthest person's reflections are much weaker. WiTrack2.0 repeats interference cancellation for the fourth person, and determines that the SNR of the maximum reflector in the resulting heatmap does not pass a threshold test. Hence, it determines that there are only four people in the scene.

We note that each of these heatmaps are scaled so that the highest power is always in red and the lowest power is in navy blue; this change in scale emphasizes the location of the strongest reflectors and allows us to better visualize their locations. To gain more insight into the power values and to better understand how SSC improves our detection of further away users, Fig. 8 plots the Signal to Interference and Noise Ratio (SINR) of the fourth person during each iteration of SSC. The fourth user's SINR initially starts at -21dB and is not visible in Fig. 6(a). Once the first and second users are removed by SSC, the SINR increases to -7dB and we can start detecting the user's presence in the back of Fig. 6(c). Performing another iteration raises the fourth person's SINR above the noise floor to 7dB. It also brings it above our threshold of 6dB – i.e., twice the noise floor – making him detectable.

We perform four additional steps to improve SSC:

- *Refocusing Step:* After obtaining the initial estimates of the locations of all four persons, WiTrack2.0 performs a focusing step for each user to refine his location estimate. This is done by reconstructing an interference-free 2D heatmap only using the range in the TOF profiles that corresponds to TOFs between TOF_{min} and TOF_{max} for that Tx-Rx pair. Figs. 6(e)- 6(h) show the images obtained from this focusing step. In these images, the location of each person is much clearer,[3] which enables higher-accuracy localization.
- *Leveraging Motion Continuity:* After obtaining the estimates from SSC, WiTrack2.0 applies a Kalman filter and performs outlier rejection to reject impractical jumps in location estimates that would otherwise correspond to abnormal human motion over a very short period of time.
- *Disentangling Crossing Paths:* To disentangle multiple people who cross paths, we look at their direction of motion before they crossed paths and project how they would proceed with the same speed and direction as they are crossing paths. This helps us with associating each person with his own trajectory after crossing. Fig. 9 shows an example with two people crossing paths and how we were able to track their trajectories despite that. Of course, this approach does not generalize to every single case, which may lead to some association errors after the crossings but not to localization errors.
- *Extending SSC to 3D Gesture Recognition:* Similar to past work [3], WiTrack2.0 can differentiate a hand motion from a whole-body motion (like walking) by leveraging the fact that a person's hand has a much smaller reflective surface than his entire body. Unlike past work, however, WiTrack2.0 can track gestures even when they are simultaneously performed by multiple users. Specifically, by exploiting SSC focusing, it zooms onto each user individually to track his gestures. In our evaluation, we focus on testing a pointing gesture, where different users point in different directions at the same time. By tracking the trajectory of each moving hand, we can determine its pointing direction. Note that we perform these pointing gestures in 3D and track hand motion by using the TOFs from the different Tx-Rx pairs to construct a 3D point cloud rather than a 2D heatmap.[4] The results in §10.3 show that we can accurately track hand gestures performed by multiple users in 3D space.

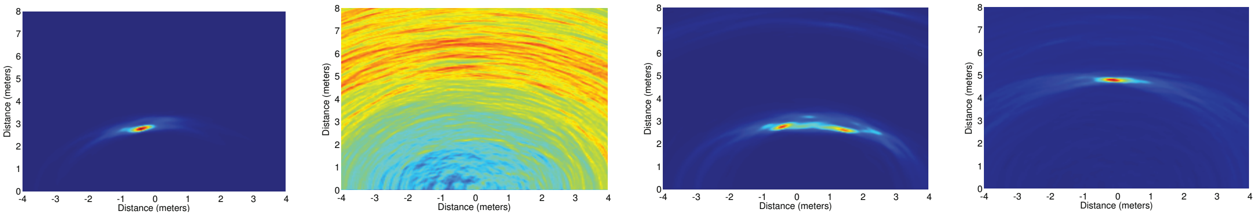
7 Localization Based on Breathing

We extend WiTrack2.0's SSC algorithm to localize static people based on their breathing. Recall from §3 that in order to track a user based on her radio reflections, we need to eliminate reflections off all static objects in the environment (like walls and furniture). This is typically achieved by performing a background subtraction step, i.e., by taking TOF profiles from adjacent time windows and subtracting them out from each other.[5]

[3] This is because all other users' reflections are eliminated, while, without refocusing, only users detected in prior iterations are eliminated.

[4] Recall from §3 that a given TOF maps to an ellipse in 2D and an ellipsoid in 3D. The intersection of ellipsoids in 3D allow us to track these pointing gestures.

[5] Recall that we obtain one TOF profile by taking an FFT over the received FMCW signal in baseband. Since the FMCW signal is repeatedly swept, we can compute a new TOF profile from each sweep.



(a) **Short subtraction window localizes a walking person.** (b) **Short subtraction window misses a static person.** (c) **Long subtraction window smears a walking person.** (d) **Long subtraction window localizes a static person.**

Figure 10—Need For Multiple Subtraction Windows. The 2D heatmaps show that a short subtraction window accurately localizes a pacing person in (a) but not a static person in (b). A long subtraction window smears the walking person's location in (c) but localizes a breathing person in (d).

Whereas this approach enables us to track moving people, it prevents us from detecting a static person – e.g., someone who is standing or sitting still. Specifically, because a static person remains in the same location, his TOF does not change, and hence his reflections would appear as static and will be eliminated in the process of background subtraction. To see this in practice, we run two experiments where we perform background subtraction by subtracting two TOF profiles that are 12.5 milliseconds apart from each other. The first experiment is performed with a walking person and the resulting heatmap is shown in Fig. 10(a), whereas the second experiment is performed in the presence of a person who is sitting at $(0,5)$ and the resulting heatmap is shown in Fig. 10(b). These experiments show how the heatmap of a moving person after background subtraction would allow us to localize him accurately, whereas the heatmap of the static person after background subtraction is very noisy and does not allow us to localize the person.

To localize static people, one needs to realize that even a static person moves slightly due to breathing. Specifically, during the process of breathing, the human chest moves by a sub-centimeter distance over a period of few seconds. The key challenge is that this change does not translate into a discernible change in the TOF of the person. However, over an interval of time of a few seconds (i.e., as the person inhales and exhales), it would result in discernible changes in the reflected signal. Therefore, by subtracting frames in time that are few seconds apart, we should be able to localize the breathing motion.

In fact, Fig. 10(d) shows that we can accurately localize a person who is sitting still by using a subtraction window of 2.5 seconds. Note, however, that this long subtraction window will introduce errors in localizing a pacing person. In particular, since typical indoor walking speed is around 1 m/s [8], subtracting two frames that are 2.5 seconds apart would result in smearing the person's location and may also result in mistaking him for two people as shown in Fig. 10(c).

Thus, to accurately localize both static and moving people, WiTrack2.0 performs background subtraction with different subtraction windows. To localize moving users, it uses a subtraction window of 12.5 ms. On the other hand, normal adults inhale and exhale over a period of 3–6 seconds [32] causing their TOF profiles to change over such intervals of time. Hence, we consider the first TOF profile during each 10-second interval, and subtract it from all subsequent TOF profiles during that interval. As a result, breathing users' reflections pop up at different instances, allowing us to detect and localize them.

8 Implementation

We built WiTrack2.0 using a single FMCW radio whose signal is fed into multiple antennas. The FMCW radio generates a low-power (sub-milliWatt) signal that sweeps 5.46-7.25 GHz every 2.5 milliseconds. The range and power are chosen in compliance with FCC regulations for consumer electronics [2].

The schematic in Fig. 5 shows how we use this radio to implement Multi-shift FMCW. Specifically, the generated sweep is delayed before being fed to directional antennas for transmission.[6] At the receive side, the signal from each receive antenna is mixed with the FMCW signal and the resulting signal is fed to the USRP. The USRP samples the signals at 2 MHz and transfers the digitized samples to the UHD driver. These samples are processed in software to localize users and recognize their gestures.[7]

The analog FMCW radio and all the USRPs are driven by the same external clock. This ensures that there is no frequency offset between their oscillators, and hence enables subtracting frames that are relatively far apart in time to enable localizing people based on breathing.

9 Evaluation

Human Subjects. We evaluate the performance of WiTrack2.0 by conducting experiments in our lab with

[6] The most straightforward option to delay the signal is to insert a wire. However, wires attenuate the signal and introduce distortion over the wide bandwidth of operation of our system, reducing its SNR. Instead, we exploit the fact that, in FMCW, time and frequency are linearly related; hence, a shift τ in time can be achieved through a shift $\Delta f = slope \times \tau$ in the frequency domain. Hence, we achieve this delay by mixing FMCW with signals whose carrier frequency is Δf. This approach also provides us with the flexibility of tuning multi-shift FMCW for different TOF_{limit}'s by simply changing these carrier frequencies.

[7] Complexity-wise, WiTrack2.0's algorithms are linear in the number of users, the number of Tx antennas, and the number of Rx antennas.

eleven human subjects: four females and seven males. The subjects differ in height from 165–185 cm as well as in weight and build and span 20 to 50 years of age. In each experiment, each subject is allowed to move as they wish throughout the room. These experiments were approved by MIT IRB protocol #1403006251.

Ground Truth. We use the VICON motion capture system to provide us with ground truth positioning information [1]. It consists of an array of infrared cameras that are fitted to the ceiling of a 5 $m \times 7$ m room, and requires instrumenting any tracked object with infrared-reflective markers. When an instrumented object moves, the system tracks the infrared markers on that object and fits them into a 3D model to identify the object's location.

We evaluate WiTrack2.0's accuracy by comparing it to the locations provided by the VICON system. To track a user using the VICON, we ask him/her to wear a hard hat that is instrumented with five infrared markers. In addition, for the gestures experiments, we ask each user to wear a glove that is instrumented with six markers.

Experimental Setup. We evaluate WiTrack2.0 in a standard office environment that has standard furniture: tables, chairs, boards, computers, etc. We experiment with two setups: line-of-sight and through-the-wall. In the through-wall experiments, WiTrack2.0 is placed outside the VICON room with all transmit and receive antennas facing one of the walls of the room. Recall that WiTrack2.0's antennas are directional and hence this setting means that the radio beam is directed toward the room. The VICON room has no windows; it has 6-inch hollow walls supported by steel frames, which is a standard setup for office buildings. In the line-of-sight experiments, we move WiTrack2.0 inside the room. In all of these experiments, the subjects' locations are tracked by both the VICON system and WiTrack2.0.

Detection. Recall that WiTrack2.0 adopts iterative cancellation to detect different users in the scene. This limits the number of users it can detect because of residual interference from previous iterations. Therefore, we run experiments to identify the maximum number of people that WiTrack2.0 can reliably detect under various conditions. Detection accuracy is measured as the percentage of time that WiTrack2.0 correctly outputs the number of users present in the environment. The number of users in each experiment is known and acts as the ground truth. We run ten experiments for each of our testing scenarios, and plot the accuracies for each them in Fig. 11.

We make two observations from this figure. First, the accuracy of detection is higher in line-of-sight than in through-wall settings. This is expected because the wall causes significant attenuation and hence reduces the SNR of the reflected signals. Second, the detection accuracy for breathing-based localization is higher than that of the tracking experiments. While this might seem surprising, it is actually due to the fact that the breathing experiments operate over longer subtraction windows. Specifically, the system outputs the number of people detected and their locations by analyzing the trace over windows of 10 seconds. In contrast, the tracking experiments require outputting a location of each person once every 12.5 ms,[8] and hence they might not be able to detect each person within such a small time window.

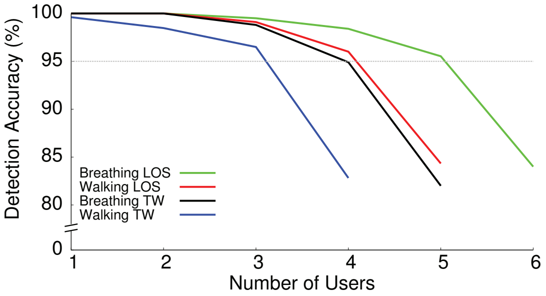


Figure 11—WiTrack2.0 's Detection Accuracy. The figure shows the percentage of time that WiTrack2.0 accurately determines the number of people in each of our evaluation scenarios.

For our evaluation of localization accuracy, we run experiments with the maximum number of people that are reliably detectable, where "reliably detectable" is defined as detected an accuracy of 95% or higher. For reference, we summarize these numbers in the table below.

	Line-of-Sight	Through-Wall
Motion Tracking	4	3
Breathing-based Localization	5	4

Table 1—Maximum Number of People Detected Reliably.

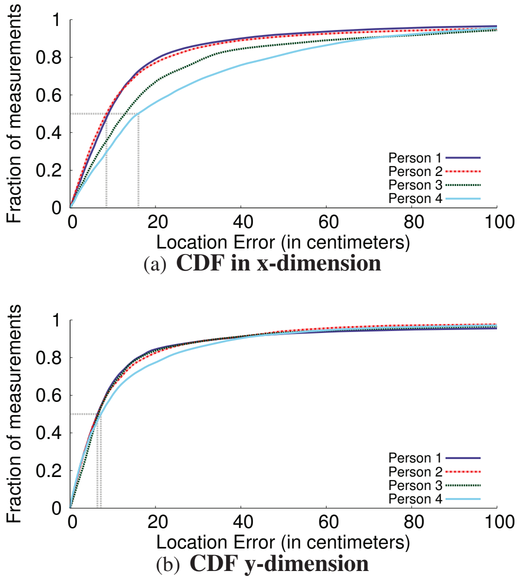
10 Performance Results

10.1 Accuracy of Multi-Person Motion Tracking

We first evaluate WiTrack2.0's accuracy in multi-person motion tracking. We run 100 experiments in total, half of them in line-of-sight and the second half in through-wall settings. In each experiment, we ask one, two, three, or four human subjects to wear the hard hats that are instrumented with VICON markers and move inside the VICON-instrumented room. Each subject's location is tracked by both the VICON system and WiTrack2.0, and each experiment lasts for one minute. Since each FMCW sweep lasts for 2.5ms and we average 5 sweeps to obtain each TOF measurement, we collect around 5,000 location readings per user per experiment.

Figs. 12 and 13 plot the CDFs of the location error along the x and y coordinates for each of the localized persons in both line-of-sight and through-wall scenarios. The subjects are ordered from the first to the last as detected by SSC. The figures reveal the following findings:

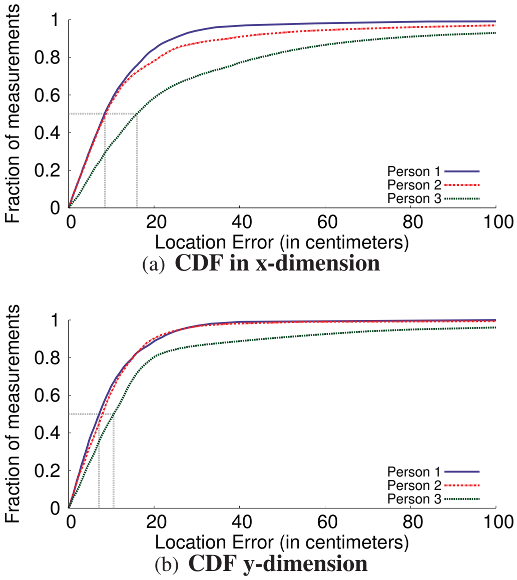
[8] Since the user is moving, combining measurements over a longer interval smears his signal.



(a) **CDF in x-dimension**

(b) **CDF y-dimension**

Figure 12—Performance of WiTrack2.0's LOS Tracking. (a) and (b) show the CDFs of the location error in x and y for each of the tracked users in LOS. Subjects are ordered from first to last detected by SSC.



(a) **CDF in x-dimension**

(b) **CDF y-dimension**

Figure 13—Performance of WiTrack2.0's Through-Wall Tracking. (a) and (b) show the CDFs of the location error in x and y for each of the tracked users. Subjects are ordered from first to last detected by SSC.

- WiTrack2.0 can accurately track the motion of four users when it is in the same room as the subjects. Its median location error for these experiments is 8.5 cm in x and 6.4 cm in y for the first user detected, and decreases to 15.9 cm in x and 7.2 cm in y for the last detected user.
- In through-wall scenarios, WiTrack2.0 can accurately localize up to three users. Its median location error for these experiments is 8.4 cm and 7.1 cm in x/y for the first detected user, and decreases to 16.1 cm and 10.5 cm in x/y for the last detected user. As expected, the accuracy when the device is placed in the same room as the users is better than when it is placed behind the wall due to the extra attenuation (reduced SNR) caused by the wall.
- The accuracy in the y dimension is better than the accuracy in the x dimension. This discrepancy is due to the asymmetric nature of WiTrack2.0's setup, where all of its antennas are arranged along the $y = 0$ axis.
- The localization accuracy decreases according to the order the SSC algorithm localizes the users. This is due to multiple reasons: First, a user detected in later iterations is typically further from the device, and hence has lower SNR, which leads to lower accuracy. Second, SSC may not perfectly remove the reflections of other users in the scene, which leads to residual interference and hence lower accuracy.

10.2 Accuracy of Breathing-based Localization

We evaluate WiTrack2.0's accuracy in localizing static people based on their breathing. We run 100 experiments in total with up to five people in the room. Half of these experiments are done in line-of-sight and the other half are through-wall. Experiments last for 3-4 minutes. Subjects wear hard hats and sit on chairs in the VICON room.

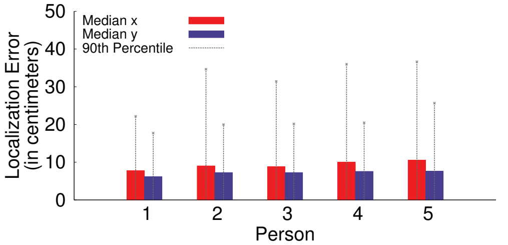


Figure 14—Accuracy for Localizing Breathing People in Line-of-Sight.. The figure shows show the median and 90^{th} percentile errors in x/y location. Subjects are ordered from first to last detected by SSC.

Figs. 14 and 15 plot WiTrack2.0's localization error in line-of-sight and through-wall settings as a function of the order with which the subject is detected by the SSC algorithm. The figures show the median and 90^{th} percentile of the estimation error for the x and y coordinates of each of the subjects. The figures show the following results:

- WiTrack2.0's breathing-based localization accuracy goes from a median of 7.24 and 6.3 cm in x/y for the nearest user to 18.31 and 10.85 cm in x/y for the furthest user, in both line-of-sight and through-wall settings.
- Localization based on breathing is more accurate than during motion. This is because when people are static, they remain in the same position, providing us with a larger number of measurements for the same location.

10.3 Accuracy of 3D Pointing Gesture Detection

We evaluate WiTrack2.0's accuracy in tracking 3D pointing gestures. We run 100 experiments in total with one to three subjects. In each of these experiments, we ask each subject to wear a glove that is instrumented with

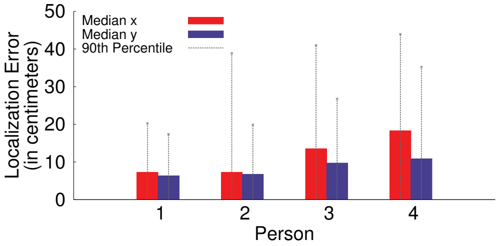
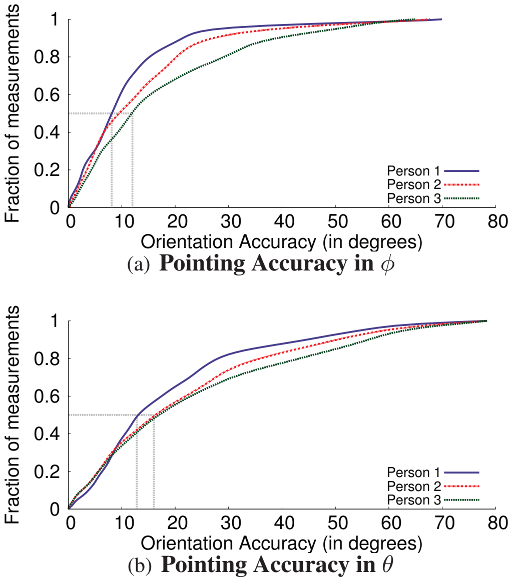


Figure 15—Accuracy for Localizing Breathing People in Through-Wall Experiments.. The figure shows show the median and 90th percentile errors in x/y location. Subjects are ordered from first to last detected by SSC.



(a) **Pointing Accuracy in ϕ**

(b) **Pointing Accuracy in θ**

Figure 16—3D Gesture Accuracy. The figure shows the CDFs of the orientation accuracy for the pointing gestures of each participant. Subjects are ordered from first to last detected by the SSC algorithm.

infrared-reflective markers, stand in a different location in the VICON room, and point his/her hand in a random 3D direction of their choice – as if they were playing a shooting game or pointing at some household appliance to control it. In most of these experiments, all subjects were performing the pointing gestures simultaneously.

Throughout these experiments, we track the 3D location of the hand using the VICON system and WiTrack2.0. We then regress on the 3D trajectory to determine the direction in which each user pointed (similar to [3]). Fig. 16(a) and 16(b) plot the CDFs of the orientation error between the angles as measured by WiTrack2.0 and the VICON for the 1st, 2nd and 3rd participant (in the order of detection by SSC). Note that we decompose the 3D pointing gesture along two directions: azimuthal (in the $x-y$ plane), which we denote as ϕ, and elevation (in the $r-z$ plane), which we denote as θ. The accuracy along both of these angles is important since appliances which the user may want to control in a home environment (e.g., lamps, screens, shades) span the 3D space.

The figure shows that the median orientation error in ϕ goes from 8.2 degrees to 12.4 degrees from the first to the third person, and from 12 degrees to 16 degrees in θ. Note that WiTrack2.0's accuracy in ϕ is slightly higher than its accuracy in θ. This is due to WiTrack2.0's setup, where the antennas are more spread out along the x than along the z, naturally leading to lower robustness to errors along the z axis, and hence lower accuracy in θ. These experiments demonstrate that WiTrack2.0 can achieve high accuracy in 3D tracking of a pointing gesture.

11 Discussion & Limitations

WiTrack2.0 marks an important step toward enabling accurate indoor localization that does not require users to hold or wear any wireless device. WiTrack2.0, however, has some limitations that are left for future work.

1. *Number of Users:* WiTrack2.0 can accurately track up to 4 moving users and 5 static users. These numbers may be sufficient for in-home tracking. However, it is always desirable to scale the system to track more users.

2. *Coverage Area:* WiTrack2.0's range is limited to 10m due to its low power. To cover larger areas and track more users, one may deploy multiple devices and hand off the trajectory tracking from one to the next, as the person moves around. Managing such a network of devices, coordinating their hand-off, and arbitrating their medium access are interesting problems to explore.

3. *Lack of Identification:* The system can track multiple users simultaneously, but it cannot identify them. Additionally, it can track limb motion (e.g., a hand) but cannot differentiate between different body parts (a hand vs. a leg). We believe that future work can investigate this issue by identifying fingerprints of various reflectors.

4. *Limited Gesture Interface:* WiTrack2.0 focuses on tracking pointing gestures; however, the user cannot move other body parts while performing the pointing gesture. Extending the system to enable rich gesture-based interfaces is an interesting avenue for future work.

Overall, we believe WiTrack2.0 pushes the limits of indoor localization and enriches the role it can play in our daily lives. By enabling smart environments to accurately follow our trajectories, it paves way for these environments to learn our habits, react to our needs, and enable us to control the Internet of Things that revolves around our networked homes and connected environments.

Acknowledgments: We thank the NETMIT group, the reviewers, and our shepherd Kate Lin for their insightful comments. We also thank Chen-Yu Hsu for helping with the experimental setup. This research is supported by NSF. The Microsoft Research PhD Fellowship has supported Fadel Adib. We thank members of the MIT Center for Wireless Networks and Mobile Computing: Amazon.com, Cisco, Google, Intel, Mediatek, Microsoft, and Telefonica for their interest and support.

References

[1] VICON T-Series. http://www.vicon.com. VICON.

[2] *Understanding the FCC Regulations for Low-power, Non-licensed Transmitters*. Office of Engineering and Technology Federal Communications Commission, 1993.

[3] F. Adib, Z. Kabelac, D. Katabi, and R. C. Miller. 3D Tracking via Body Radio Reflections. In *Usenix NSDI*, 2014.

[4] F. Adib and D. Katabi. See through walls with Wi-Fi! In *ACM SIGCOMM*, 2013.

[5] P. Bahl and V. N. Padmanabhan. Radar: An in-building RF-based user location and tracking system. In *IEEE INFOCOM*, 2000.

[6] D. Bharadia, E. McMilin, and S. Katti. Full duplex radios. In *SIGCOMM*. ACM, 2013.

[7] M. Bocca, O. Kaltiokallio, N. Patwari, and S. Venkatasubramanian. Multiple target tracking with RF sensor networks. *Mobile Computing, IEEE Transactions on*, 2013.

[8] R. Bohannon. Comfortable and maximum walking speed of adults aged 20-79 years: reference values and determinants. *Age and ageing*, 1997.

[9] P. K. Chan, W. Jin, J. Gong, and N. Demokan. Multiplexing of fiber bragg grating sensors using a fmcw technique. *IEEE Photonics Technology Letters*, 1999.

[10] G. Charvat, L. Kempel, E. Rothwell, C. Coleman, and E. Mokole. An ultrawideband (UWB) switched-antenna-array radar imaging system. In *IEEE ARRAY*, 2010.

[11] K. Chintalapudi, A. Padmanabha Iyer, and V. N. Padmanabhan. Indoor localization without the pain. In *ACM MobiCom*. ACM, 2010.

[12] S. Gollakota and D. Katabi. Zigzag decoding: combating hidden terminals in wireless networks. In *ACM SIGCOMM*, 2008.

[13] S. Hantscher, A. Reisenzahn, and C. Diskus. Through-wall imaging with a 3-d uwb sar algorithm. *Signal Processing Letters, IEEE*, 2008.

[14] J. Högbom. Aperture synthesis with a non-regular distribution of interferometer baselines. *Astronomy and Astrophysics Supplement Series*, 1974.

[15] Y. Huang, P. V. Brennan, D. Patrick, I. Weller, P. Roberts, and K. Hughes. Fmcw based mimo imaging radar for maritime navigation. *Progress In Electromagnetics Research*, 2011.

[16] Y. Jia, L. Kong, X. Yang, and K. Wang. Through-wall-radar localization for stationary human based on life-sign detection. In *IEEE RADAR*, 2013.

[17] K. Joshi, S. Hong, and S. Katti. Pinpoint: Localizing interfering radios. In *Usenix NSDI*, 2013.

[18] S. Nannuru, Y. Li, Y. Zeng, M. Coates, and B. Yang. Radio-frequency tomography for passive indoor multitarget tracking. *Mobile Computing, IEEE Transactions on*, 2013.

[19] N. Patwari, L. Brewer, Q. Tate, O. Kaltiokallio, and M. Bocca. Breathfinding: A wireless network that monitors and locates breathing in a home. *Selected Topics in Signal Processing, IEEE Journal of*, 2014.

[20] Q. Pu, S. Jiang, S. Gollakota, and S. Patel. Whole-home gesture recognition using wireless signals. In *ACM MobiCom*, 2013.

[21] A. Rai, K. K. Chintalapudi, V. N. Padmanabhan, and R. Sen. Zee: zero-effort crowdsourcing for indoor localization. In *ACM MobiCom*, 2012.

[22] S. Rallapalli, A. Ganesan, K. Chintalapudi, V. N. Padmanabhan, and L. Qiu. Enabling physical analytics in retail stores using smart glasses. In *ACM MobiCom*, 2014.

[23] T. Ralston, G. Charvat, and J. Peabody. Real-time through-wall imaging using an ultrawideband multiple-input multiple-output (MIMO) phased array radar system. In *IEEE ARRAY*, 2010.

[24] A. Saeed, A. Kosba, and M. Youssef. Ichnaea: A low-overhead robust wlan device-free passive localization system. *Selected Topics in Signal Processing, IEEE Journal of*, 2014.

[25] M. Seifeldin, A. Saeed, A. Kosba, A. El-Keyi, and M. Youssef. Nuzzer: A large-scale device-free passive localization system for wireless environments. *Mobile Computing, IEEE Transactions on*, 2013.

[26] S. Sen, B. Radunovic, R. R. Choudhury, and T. Minka. Spot localization using phy layer information. In *ACM MobiSys*, 2012.

[27] P. Setlur, G. Alli, and L. Nuzzo. Multipath exploitation in through-wall radar imaging via point spread functions. *Image Processing, IEEE Transactions on*, 2013.

[28] G. E. Smith and B. G. Mobasseri. Robust through-the-wall radar image classification using a target-model alignment procedure. *Image Processing, IEEE Transactions on*, 2012.

[29] A. R. Thompson, J. M. Moran, and G. W. Swenson Jr. *Interferometry and synthesis in radio astronomy*. John Wiley & Sons, 2008.

[30] J. Tsao and B. D. Steinberg. Reduction of sidelobe and speckle artifacts in microwave imaging: the clean technique. *Antennas and Propagation, IEEE Transactions on*, 1988.

[31] A. Vasilyev. *The optoelectronic swept-frequency laser and its applications in ranging, three-dimensional imaging, and coherent beam combining of chirped-seed amplifiers*. PhD thesis, 2013.

[32] H. K. Walker, W. D. Hall, and J. W. Hurst. *Respiratory Rate and Pattern*. Butterworths, 1990.

[33] G. Wang, Y. Zou, Z. Zhou, K. Wu, and L. M. Ni. We can hear you with wi-fi! In *ACM MobiCom*, 2014.

[34] J. Wang and D. Katabi. Dude, where's my card? RFID positioning that works with multipath and non-line of sight. In *ACM SIGCOMM*, 2013.

[35] Y. Wang, J. Liu, Y. Chen, M. Gruteser, J. Yang, and H. Liu. E-eyes: device-free location-oriented activity identification using fine-grained wifi signatures. In *ACM MobiCom*. ACM, 2014.

[36] J. Wilson and N. Patwari. Radio tomographic imaging with wireless networks. In *IEEE Transactions on Mobile Computing*, 2010.

[37] P. W. Wolniansky, G. J. Foschini, G. Golden, and R. Valenzuela. V-blast: An architecture for realizing very high data rates over the rich-scattering wireless channel. In *IEEE ISSSE*, 1998.

[38] J. Xiong and K. Jamieson. ArrayTrack: a fine-grained indoor location system. In *Usenix NSDI*, 2013.

[39] Y. Xu, S. Wu, C. Chen, J. Chen, and G. Fang. A novel method for automatic detection of trapped victims by ultrawideband radar. *Geoscience and Remote Sensing, IEEE Transactions on*, 2012.

[40] M. Youssef, M. Mah, and A. Agrawala. Challenges: device-free passive localization for wireless environments. In *ACM MobiCom*, 2007.

Making Sense of Performance in Data Analytics Frameworks

Kay Ousterhout*, Ryan Rasti*†◇, Sylvia Ratnasamy*, Scott Shenker*†, Byung-Gon Chun‡
*UC Berkeley, †ICSI, ◇VMware, ‡Seoul National University

Abstract

There has been much research devoted to improving the performance of data analytics frameworks, but comparatively little effort has been spent systematically identifying the performance bottlenecks of these systems. In this paper, we develop blocked time analysis, a methodology for quantifying performance bottlenecks in distributed computation frameworks, and use it to analyze the Spark framework's performance on two SQL benchmarks and a production workload. Contrary to our expectations, we find that (i) CPU (and not I/O) is often the bottleneck, (ii) improving network performance can improve job completion time by a median of at most 2%, and (iii) the causes of most stragglers can be identified.

1 Introduction

Large-scale data analytics frameworks such as Hadoop [13] and Spark [51] are now in widespread use. As a result, both academia and industry have dedicated significant effort towards improving the performance of these frameworks.

Much of this performance work has been motivated by three widely-accepted mantras about the performance of data analytics:

1. **The network is a bottleneck.** This has motivated work on a range of network optimizations, including load balancing across multiple paths, leveraging application semantics to prioritize traffic, aggregating data to reduce traffic, isolation, and more [6, 14, 17–21, 27, 28, 41, 42, 48, 53].
2. **The disk is a bottleneck.** This has led to work on using the disk more efficiently [43] and caching data in memory [9, 30, 47, 51].
3. **Straggler tasks significantly prolong job completion times and have largely unknown underlying causes.** This has driven work on mitigation using task speculation [8, 10, 11, 52] or running shorter tasks to improve load balancing [39]. Researchers have been able to identify and target a small number of underlying causes such as data skew [11, 26, 29] and popularity skew [7].

Most of this work focuses on a particular aspect of the system in isolation, leaving us without a comprehensive understanding of which factors are most important to the end-to-end performance of data analytics workloads.

This paper makes two contributions towards a more comprehensive understanding of performance. First, we develop a methodology for analyzing end-to-end performance of data analytics frameworks; and second, we use our methodology to study performance of two SQL benchmarks and one production workload. Our results run counter to all three of the aforementioned mantras.

The first contribution of this paper is *blocked time analysis*, a methodology for quantifying performance bottlenecks. Identifying bottlenecks is challenging for data analytics frameworks because of pervasive parallelism: jobs are composed of many parallel tasks, and each task uses pipelining to parallelize the use of network, disk, and CPU. One task may be bottlenecked on different resources at different points in execution, and at any given time, tasks for the same job may be bottlenecked on different resources. Blocked time analysis uses extensive white-box logging to measure how long each task spends blocked on a given resource. Taken alone, these per-task measurements allow us to understand straggler causes by correlating slow tasks with long blocked times. Taken together, the per-task measurements for a particular job allow us to simulate how long the job would have taken to complete if the disk or network were infinitely fast, which provides an upper bound on the benefit of optimizing network or disk performance.

The second contribution of this paper is using blocked time analysis to understand Spark's performance on two industry benchmarks and one production workload. In studying the applicability of the three aforementioned claims to these workloads, we find:

1. **Network optimizations can only reduce job completion time by a median of at most 2%.** The network is not a bottleneck because much less data is sent over the network than is transferred to and from disk. As a result, network I/O is mostly irrelevant to overall performance, even on 1Gbps networks.
2. **Optimizing or eliminating disk accesses can only reduce job completion time by a median of at most 19%.** CPU utilization is typically much higher than disk utilization; as a result, engineers should be careful about trading off I/O time for CPU time by, for example, using more sophisticated serialization and compression techniques.
3. **Optimizing stragglers can only reduce job completion time by a median of at most 10%, and in**

Workload name	Total queries	Cluster size	Data size
Big Data Benchmark [46], Scale Factor 5 (BDBench)	50 (10 unique queries, each run 5 times)	40 cores (5 machines)	60GB
TPC-DS [45], Scale Factor 100	140 (7 users, 20 unique queries)	160 cores (20 machines)	17GB
TPC-DS [45], Scale Factor 5000	260 (13 users, 20 unique queries)	160 cores (20 machines)	850GB
Production	30 (30 unique queries)	72 cores (9 machines)	tens of GB

Table 1: Summary of workloads run. We study one larger workload in §6.

75**% of queries, we can identify the cause of more than** 60**% of stragglers.** Blocked-time analysis illustrates that the two leading causes of Spark stragglers are Java's garbage collection and time to transfer data to and from disk. We found that targeting the underlying cause of stragglers could reduce non-straggler runtimes as well, and describe one example where understanding stragglers in early experiments allowed us to identify a bad configuration that, once fixed, reduced job completion time by a factor of two.

These results question the prevailing wisdom about the performance of data analytics frameworks. By necessity, our study does not look at a vast range of workloads nor a wide range of cluster sizes, because the ability to add finer-grained instrumentation to Spark was critical to our analysis. As a result, we cannot claim that our results are broadly representative. However, the fact that the prevailing wisdom about performance is so incorrect on the workloads we do consider suggests that there is much more work to be done before our community can claim to understand the performance of data analytics frameworks.

To facilitate performing blocked time analysis on a broader set of workloads, we have added almost all[1] of our instrumentation to Spark and made our analysis tools publicly available. We have also published the detailed benchmark traces that we collected so that other researchers may reproduce our analysis or perform their own [37].

The remainder of this paper begins by describing blocked time analysis and the associated instrumentation (§2). Next, we explore the importance of disk I/O (§3), the importance of network I/O (§4), and the importance and causes of stragglers (§5); in each of these sections, we discuss the relevant related work and contrast it with our results. We explore the impact of cluster and data size on our results in §6. We end by arguing that future system designs should consider performance measurement as a first-class concern (§7).

2 Methodology

This section describes the workloads we ran, the blocked time analysis we used to understand performance, and our experimental setup.

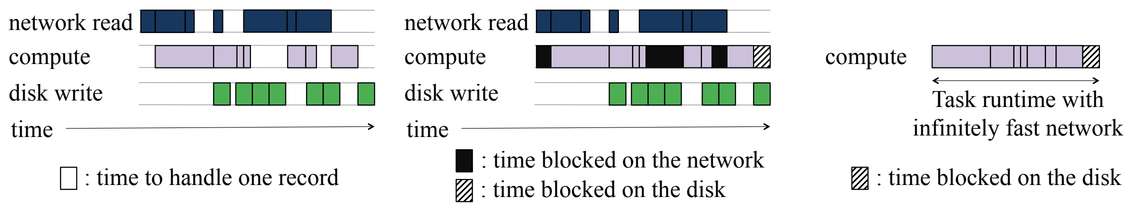
2.1 Workloads

Our analysis centers around fine-grained instrumentation of two benchmarks and one production workload running on Spark, summarized in Table 1.

The big data benchmark (BDBench) [46] was developed to evaluate the differences between analytics frameworks and was derived from a benchmark developed by Pavlo et al. [40]. The input dataset consists of HTML documents from the Common Crawl document corpus [2] combined with SQL summary tables generated using Intel's Hadoop benchmark tool [50]. The benchmark consists of four queries including two exploratory SQL queries, one join query, and one page-rank-like query. The first three queries have three variants that each use the same input data size but have different result sizes to reflect a spectrum between business-intelligence-like queries (with result sizes that could fit in memory on a business intelligence tool) and ETL-like queries with large result sets that require many machines to store. We run the queries in series and run five iterations of each query. We use the same configuration that was used in published results [46]: we use a scale factor of five (which was designed to be run on a cluster with five worker machines), and we run two versions of the benchmark. The first version operates on data stored in-memory using SparkSQL's columnar cache (cached data is not replicated) and the second version operates on data stored on-disk using Hadoop Distributed File System (HDFS), which triply replicates data for fault-tolerance.

Our second benchmark is a variant of the Transaction Processing Performance Council's decision-support benchmark (TPC-DS) [45]. The TPC-DS benchmark was designed to model multiple users running a variety of decision-support queries including reporting, interactive OLAP, and data mining queries. All of the users run in parallel; each user runs the queries in series in a random order. The benchmark models data from a retail product supplier about product purchases. We use a subset of 20 queries that was selected in an existing industry benchmark that compares four analytics frameworks [25]. Similar to with the big data benchmark, we run two variants. The first variant stores data on-disk using Parquet [1], a compressed columnar storage format that is the recommended storage format for high performance with Spark SQL, and uses a scale factor of 5000. The

[1] Some of our logging needed to be added outside of Spark, as we elaborate on in §2.3.1, because it could not be implemented in Spark with sufficiently low overhead.


network read
compute
disk write
time
□ : time to handle one record
network read
compute
disk write
time
■ : time blocked on the network
▨ : time blocked on the disk
compute
Task runtime with
infinitely fast network
▨ : time blocked on the disk

(a) Pipelined execution of a typical Spark task (b) Blocked times that we measured (c) Task runtime without blocking on network

Figure 1: Each Spark task pipelines use of network, CPU, and disk, as shown in (a). To understand the importance of disk and network, we measure times when a task's compute thread blocks on the network or disk, as shown in (b). To determine the best-case task runtime resulting from network (or disk) optimizations, we subtract time blocked on network (or disk), as shown in (c).

second, in-memory variant uses a smaller scale factor of 100; this small scale factor is necessary because SparkSQL's cache is not well optimized for the type of data used in the TPC-DS benchmark, so while the data only takes up 17GB in the compressed on-disk format, it occupies 200GB in memory. We run both variants of the benchmark on a cluster of 20 machines.

The final Spark workload described by the results in this paper is a production workload from Databricks that uses their cloud product [3] to submit ad-hoc Spark queries. Input data for the queries includes a large fact table with over 50 columns. The workload includes a small number of ETL queries that read input data from an external file system into the memory of the cluster; subsequent queries operate on in-memory data and are business-intelligence-style queries that aggregate and summarize data. Data shown in future graphs breaks the workload into the in-memory and on-disk components. For confidentiality reasons, further details of the workload cannot be disclosed.

2.2 Framework architecture

All three workloads are SQL workloads that use Spark-SQL [4] to compile SQL queries into Spark jobs. Spark jobs are broken down into stages composed of many parallel tasks. The tasks in a stage each perform the same computation using different subsets of the stage's input data. Early stages read input data from a distributed file system (e.g., HDFS) or Spark's cache, whereas later stages typically read input data using a network shuffle, where each task reads a subset of the output data from all of the previous stage's tasks. In the remainder of this paper, we use "map task" to refer to tasks that read blocks of input data stored in a distributed file system, and "reduce task" to refer to tasks that read data shuffled from the previous stage of tasks. Each Spark job is made up of a directed acyclic graph of one or more stages. As a result, a single Spark job may contain multiple stages of reduce tasks; for example, to compute the result of a SQL query that includes multiple joins (in contrast, with MapReduce, all jobs include exactly one map and optionally one reduce).

2.3 Blocked time analysis

The goal of this paper is to understand performance of workloads running on Spark; this is a challenging goal for two reasons. First, understanding the performance of a single task is challenging because tasks use pipelining, as shown in Figure 1(a). As a result, tasks often use multiple resources simultaneously, and different resources may be the bottleneck at different times in task execution. Second, understanding performance is challenging because jobs are composed of many tasks that run in parallel, and each task in a job may have a unique performance profile.

To make sense of performance, we focus on *blocked time analysis*, which allows us to quantify how much more quickly a job would complete if tasks never blocked on the disk or the network (§3.3 explains why we cannot use blocked time analysis to understand CPU use). The resulting job completion time represents a best-case scenario of the possible job completion time after implementing a disk or network optimization. Blocked time analysis lacks the sophistication and generality of general purpose distributed systems performance analysis tools (e.g., [5, 15]), and unlike black-box approaches, requires adding instrumentation within the application. We use blocked-time analysis because unlike existing approaches, it provides a single, easy to understand number to characterize the importance of disk and network use.

2.3.1 Instrumentation

To understand the performance of a particular task, we focus on blocked time: time the task spends blocked on the network or the disk (shown in Figure 1(b)). We focus on blocked time from the perspective of the compute thread because it provides a single vantage point from which to measure. The task's computation runs as a single thread, whereas network requests are issued by multiple threads that operate in the background, and disk I/O is pipelined by the OS, outside of the Spark process. We focus on blocked time, rather than measuring all time when the task is using the network or the disk, because network or disk performance improvements cannot speed up parts of the task that execute in parallel with network or disk use.

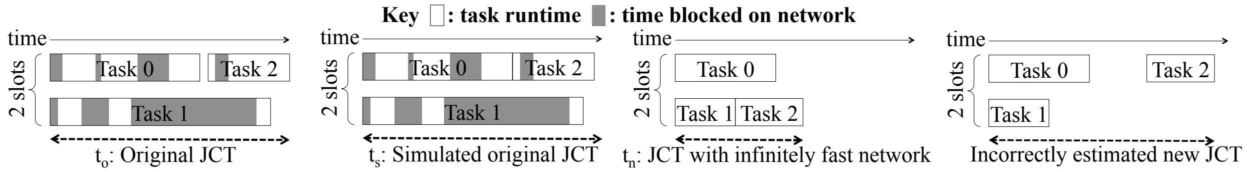

Key □: task runtime ■: time blocked on network
time
2 slots
Task 0
Task 2
Task 1
t_o: Original JCT
t_s: Simulated original JCT
t_n: JCT with infinitely fast network
Incorrectly estimated new JCT

Figure 2: To compute a job's completion time (JCT) without time blocked on network, we subtract the time blocked on the network from each task, and then simulate how the tasks would have been scheduled, given the number of slots used by the original runtime and the new task runtimes. We perform the same computation for disk.

Obtaining the measurements shown in Figure 1(b) required significant improvements to the instrumentation in Spark and in HDFS.While some of the instrumentation required was already available in Spark, our detailed performance analysis revealed that existing logging was often incorrect or incomplete [33–36, 38, 44]. Where necessary, we fixed existing logging and pushed the changes upstream to Spark.

We found that cross validation was crucial to validating that our measurements were correct. In addition to instrumentation for blocked time, we also added instrumentation about the CPU, network, and disk utilization on the machine while the task was running (per-task utilization cannot be measured in Spark, because all tasks run in a single process). Utilization measurements allowed us to cross validate blocked times; for example, by ensuring that when tasks spent little time blocked on I/O, CPU utilization was correspondingly high.

As part of adding instrumentation, we measured Spark's performance before and after the instrumentation was added to ensure the instrumentation did not add to job completion time. To ensure logging did not affect performance, we sometimes had to add logging outside of Spark. For example, to measure time spent reading input data, we needed to add logging in the HDFS client when the client reads large "packets" of data from disk, to ensure that timing calls were amortized over a relatively time-consuming read from disk.[2] Adding the logging in Spark, where records are read one at a time from the HDFS client interface, would have degraded performance.

2.3.2 Simulation

Spark instrumentation allowed us to determine how long each task was blocked on network (or disk) use; subtracting these blocked times tells us the shortest possible task runtime that would result from optimizing network (or disk) performance, as shown in Figure 1(c).[3] Next, we used a simulation to determine how the shorter task completion times would affect job completion time. The simulation replays the execution of the scheduling of the job's tasks, based on the number of slots used by the original job and the new task runtimes. Figure 2 shows a simple example. The example on the far right illustrates why we need to replay execution rather than simply use the original task completion times minus blocked time: that approach underestimates improvements because it does not account for multiple waves of tasks (task 2 should start when the previous task finishes, not at its original start time) and does not account for the fact that tasks might have been scheduled on different machines given different runtimes of earlier tasks (task 2 should start on the slot freed by task 1 completing). We replay the job based only on the number of slots used by the original job, and do not take into account locality constraints that might have affected the scheduling of the original job. This simplifying assumption does not significantly impact the accuracy of our simulation: at the median, the time predicted by our simulation is within 4% of the runtime of the original job. The ninety-fifth percentile error is at most 7% for the benchmark workloads and 27% for the production workload.[4] In order to minimize the effect of this error on our results, we always compare the simulated time without network (or disk) to the simulated original time. For example, in the example shown in Figure 2, we would report the improvement as t_n/t_s, rather than as t_n/t_o. This focuses our results on the impact of a particular resource rather than on error introduced by our simulation.

2.4 Cluster setup

For the benchmark workloads, we ran experiments using a cluster of Amazon EC2 m2.4xlarge instances, which each have 68.4GB of memory, two disks, and eight cores. Our experiments use Apache Spark version 1.2.1 and Hadoop version 2.0.2. Spark runs queries in long-running processes, meaning that production users of Spark will run queries in a JVM that has been running for a long period of time. To emulate that environment, before running each benchmark, we ran a single full trial of all of the benchmark queries to warm up the JVM. For the big data benchmark, where only one query runs at a time, we cleared the

[2] This logging is available in a modified version of Hadoop at `https://github.com/kayousterhout/hadoop-common/tree/2.0.2-instrumented`

[3] The task runtime resulting from subtracting all time blocked on the network may be lower than the runtime that would result even if the network were infinitely fast, because eliminating network blocked time might result in more time blocked on disk. As we emphasize throughout the paper, our results represent a bound on the largest possible improvement from optimizing network or disk performance.

[4] The production workload has higher error because we don't model pauses between stages that occur when SparkSQL is updating the query plan. If we modeled these pauses, the impact of disk and network I/O would be lower.

OS buffer cache on all machines before launching each query, to ensure that input data is read from disk. While our analysis focuses on one cluster size and data size for each benchmark, we illustrate that scaling to larger clusters does not significantly impact our results in §6.

The production workload ran on a 9-machine cluster with 250GB of memory; further details about the cluster configuration are proprietary. The cluster size and hardware is representative of Databricks' users.

2.5 Production traces

Where possible, we sanity-checked our results with coarse-grained analysis of traces from Facebook, Google, and Microsoft. The Facebook trace includes 54K jobs run during a contiguous period in 2010 on a cluster of thousands of machines (we use a 1-day sample from this trace). The Google data includes all MapReduce jobs run at Google during three different one month periods in 2004, 2006, and 2007, and the Microsoft data includes data from an analytics cluster with thousands of servers on a total of eighteen different days in 2009 and 2010. While our analysis would ideally have used only data from production analytics workloads, all data made available to us includes insufficient instrumentation to compute blocked time. For example, the default logs written by Hadoop (available for the Facebook cluster) include only the total time for each map task, but do not break map task time into how much time was spent reading input data and how much time was spent writing output data. This has forced researchers to rely on estimation techniques that can be inaccurate, as we show in §4.4. Therefore, our analysis begins with a detailed instrumentation of Spark, but in most cases, we demonstrate that our high-level takeaways are compatible with production data.

3 How important is disk I/O?

Previous work has suggested that reading input data from disk can be a bottleneck in analytics frameworks; for example, Spark describes speedups of 40× for generating a data analytics report as a result of storing input and output data in memory using Spark, compared to storing data on-disk and using Hadoop for computation [51]. PACMan reported reducing average job completion times by 53% as a result of caching data in-memory [9]. The assumption that many data analytics workloads are I/O bound has driven numerous research proposals (e.g., Themis [43], Tachyon [30]) and the implementation of in-memory caching in industry [47]. Based on this work, our expectation was that time blocked on disk would represent the majority of job completion time.

3.1 How much time is spent blocked on disk I/O?

Using blocked time analysis, we compute the improvement in job completion time if tasks did not spend any time blocked on disk I/O.[5] This involved measuring time blocked on disk I/O at four different points in task execution:

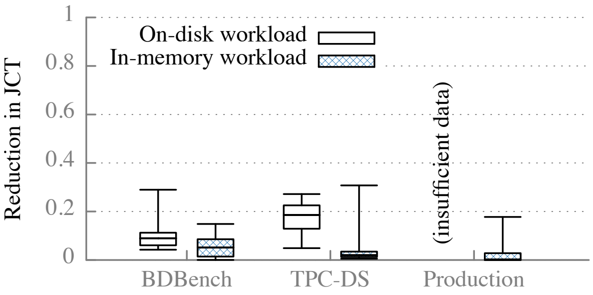
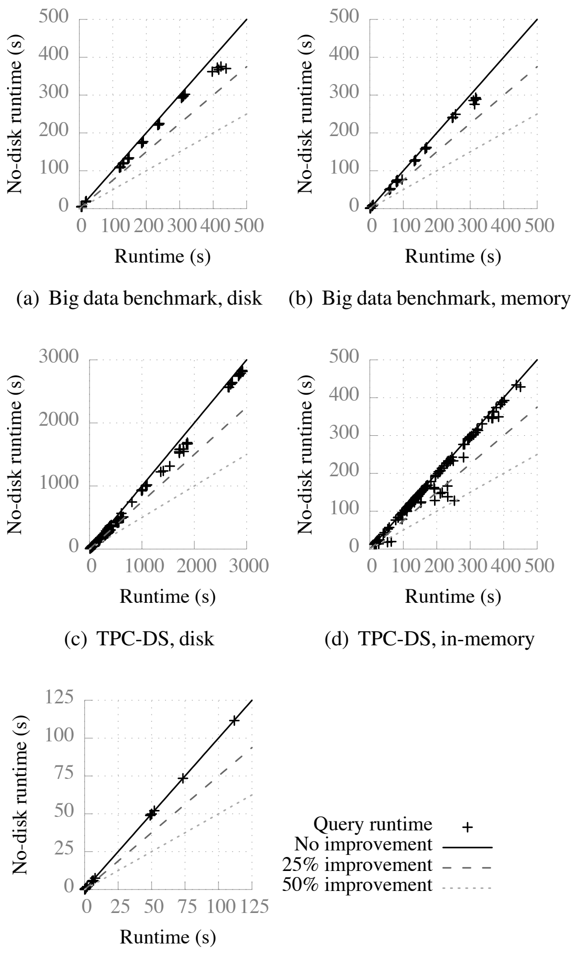


Figure 3: Improvement in job completion time (JCT) as a result of eliminating all time spent blocked on disk I/O. Boxes depict 25th, 50th, and 75th percentiles; whiskers depict 5th and 95th percentiles.

1. Reading input data stored on-disk (this only applies for the on-disk workloads; in-memory workloads read input from memory).
2. Writing shuffle data to disk. Spark writes all shuffle data to disk, even when input data is read from memory.
3. Reading shuffle data from a remote disk. This time includes both disk time (to read data from disk) and network time (to send the data over the network). Network and disk use is tightly coupled and thus challenging to measure separately; we measure the total time as an *upper bound* on the improvement from optimizing disk performance.
4. Writing output data to local disk and two remote disks (this only applies for the on-disk workloads). Again, the time to write data to remote disks includes network time as well; we measure both the network and disk time, making our results an upper bound on the improvement from optimizing disk.

Using blocked time analysis, we find that the median improvement from eliminating all time blocked on disk is at most 19% across all workloads, as shown in Figure 3. The y-axis in Figure 3 describes the relative reduction in job completion time; a reduction of 0.1 means that the job could complete 10% faster as a result of eliminating time blocked on the disk. The figure illustrates the distribution over jobs, including all trials of each job in each workload. Boxes depict 25th, 50th, and 75th percentiles; whiskers

[5] This measurement includes only time blocked on disk requests, and does not include CPU time spent deserializing byte buffers into Java objects. This time is sometimes considered disk I/O because it is a necessary side effect of storing data outside of the JVM; we consider only disk hardware performance here, and discuss serialization time separately in §3.5.



(a) Big data benchmark, disk (b) Big data benchmark, memory

(c) TPC-DS, disk (d) TPC-DS, in-memory

(e) Production, in-memory

Figure 4: Comparison of original runtimes of Spark jobs to runtimes when all time blocked on the disk has been eliminated.

depict 5th and 95th percentiles. The variance stems from the fact that different jobs are affected differently by disk. The on-disk queries in the production workload are not shown in Figure 3 because, as described in §2.3.1, instrumenting time to read input data required adding instrumentation to HDFS, which was not possible to do for the production cluster. We show the same results in Figure 4, which instead plots the absolute runtime originally and the absolute runtime once time blocked on disk has been eliminated. The scatter plots illustrate that long jobs are not disproportionately affected by time reading data from disk.

For in-memory workloads, the median improvement from eliminating all time blocked on disk is 2-5%; the fact that this improvement is non-zero is because even in-memory workloads store shuffle data on disk.

A median improvement in job completion time of 19% is a respectable improvement in runtime, but is lower than we expected for workloads that read input data and store output data on-disk. The following two subsections make more sense of this number, first by considering the effect of our hardware setup, and then by examining alternate metrics to put this number in the context of how tasks spend their time.

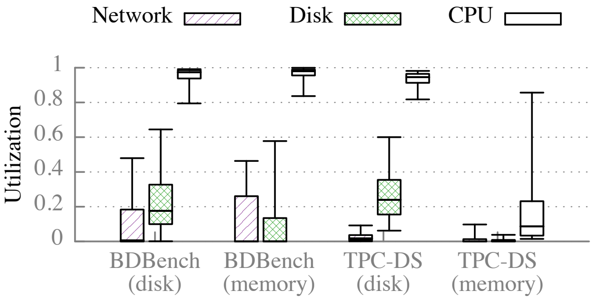


Figure 5: Average network, disk, and CPU utilization while tasks were running. CPU utilization is the total fraction of non-idle CPU milliseconds while the task was running, divided by the eight total cores on the machine. Network utilization is the bandwidth usage divided by the machine's 1Gbps available bandwidth. All utilizations are obtained by reading the counters in the /proc file system at the beginning and end of each task. The distribution is across all tasks in each workload, weighted by task duration.

3.2 How does hardware configuration affect these results?

Hardware configuration impacts blocked time: tasks run on machines with fewer disks relative to the number of CPU cores would have spent more time blocked on disk, and vice versa. In its hardware recommendations for users purchasing Hadoop clusters, one vendor recommends machines with at least a 1:3 ratio of disks to CPU cores [31]. In 2010, Facebook's Hadoop cluster included machines with between a 3:4 and 3:1 ratio of disks to CPU cores [16]. Thus, our machines, with a 1:4 ratio of disks to CPU cores, have relatively under-provisioned I/O capacity. As a result, I/O may appear *more* important in our measurements than it would in the wild, so our results on time blocked on disk represent even more of an upper bound on the importance of I/O.

The second aspect of our setup that affects results is the number of concurrent tasks run per machine; we run one task per core, consistent with the Spark default.

3.3 How does disk utilization compare to CPU utilization?

Our result that eliminating time blocked on disk I/O can only improve job completion time by a median of at most 19% suggests that jobs may be CPU bound. Unfortunately, we cannot use blocked time analysis to understand the importance of compute time, because we cannot measure when task I/O is blocked waiting for computation. The operating system often performs disk

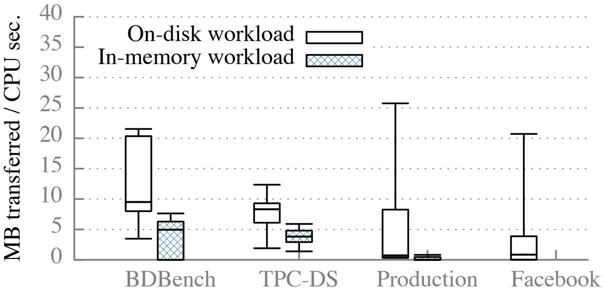


Figure 6: Average megabytes transferred to or from disk per non-idle CPU second for all jobs we ran. The median job transfers less than 10 megabytes to/from disk per CPU second; given that effective disk throughput was approximately 90 megabytes per second per disk while running our benchmarks, this demand can easily be met by the two disks on each machine.

I/O in the background, while the task is also using the CPU. Measuring when a task is using only a CPU versus when background I/O is occurring is thus difficult, and is further complicated by the fact that all Spark tasks on a single machine run in the same process.

Because we can't use blocked time analysis to understand the importance of CPU use, we instead examine CPU and disk utilization. Figure 5 plots the distribution across all tasks in the big data benchmark and TPC-DS benchmark of the CPU utilization compared to disk utilization. For this workload, the plot illustrates that on average, queries are more CPU bound than I/O bound. Hence, tasks are likely blocked waiting on computation to complete more often than they are blocked waiting for disk I/O. On clusters with more typical ratios of disk to CPU use, the disk utilization will be even lower relative to CPU utilization.

3.4 Sanity-checking our results against production traces

The fact that the disk is not the bottleneck in our workloads left us wondering whether our workloads were unusually CPU bound, which would mean that our results were not broadly representative. Unfortunately, production data analytics traces available to us do not include blocked time or disk utilization information. However, we were able to use aggregate statistics about the CPU and disk use of jobs to compare the I/O demands of our workloads to the I/O demands of larger production workloads. In particular, we measured the I/O demands of our workloads by measuring the ratio of data transferred to disk to non-idle CPU seconds:

$$\text{MB / CPU second} = \frac{\text{Total MB transferred to/from disk}}{\text{Total non-idle CPU seconds}}$$

This metric is imperfect because it looks at the CPU and disk requirements of the job as a whole, and does not account for variation in resource usage during a job. Nonetheless, it allows us to understand how the aggregate disk demands of our workloads compare to large-scale production workloads.

Using this metric, we found that the I/O demands of the three workloads we ran do not differ significantly from I/O demands of production workloads. Figure 6 illustrates that for the benchmarks and production workload we instrumented, the median MB / CPU second is less than 10. Figure 6 also illustrates results for the trace from Facebook's Hadoop cluster. The MB / CPU second metric is useful in comparing to Hadoop performance because it relies on the volume of data transferred to disk and the CPU milliseconds to characterize the job's demand on I/O, so abstracts away many inefficiencies in Hadoop's implementation (for example, it abstracts away the fact that a CPU-bound query may take much longer than the CPU milliseconds used due to inefficient resource use). The number of megabytes transferred to disk per CPU second is lower for the Facebook workload than for our workloads: the median is just 3MB/s, compared to a median of 9MB/s for the big data benchmark on-disk workload and 8MB/s for the TPC-DS workload.

We also examined aggregate statistics published about Microsoft's Cosmos cluster and Google's MapReduce cluster, shown in Tables 2 and 3. Unlike the Facebook trace, those statistics do not include a measurement of the CPU time spent by jobs, and instead quote the total time that tasks were running, aggregated across all jobs [23]. In computing the average rate at which tasks transfer data to and from disk, we assume that the input data is read once from disk, intermediate data is written once (by map tasks that generate the data) and read once (by reduce tasks that consume the data), and output data is written to disk three times (assuming the industry standard triply-replicated output data). As shown in Tables 2 and 3, based on this estimate, Google jobs transfer an average of 0.787 to 1.47 MB/s to disk, and Microsoft jobs transfer an average of 6.61 to 10.58 MB/s to disk. These aggregate numbers reflect an estimate of *average* I/O use so do not reflect tail behavior, do not include additional I/O that may have occurred (e.g., to spill intermediate data), and are not directly comparable to our results because unlike the CPU milliseconds, the total task time includes time when the task was blocked on network or disk I/O.[6] The takeaway from these results should not be the precise value of these aggregate metrics, but rather that sanity checking our results against production traces does not lead us to believe that production workloads have dramatically different

[6] For the Google traces, the aggregate numbers are skewed by the fact that, at the time, a few MapReduce jobs that consumed significant resources were also very CPU intensive (in particular, the final phase of the indexing pipeline involved significant computation). These jobs also performed some additional disk I/O from within the user-defined map and reduce functions [23]. We lack sufficient information to quantify these factors, so they are not included in our estimate of MB/s.

	Aug. '04	Mar. '06	Sep. '07
Map input data (TB) [24]	3,288	52,254	403,152
Map output data (TB) [24]	758	6,743	34,774
Reduce output data (TB) [24]	193	2,970	14,018
Task years used [24]	217	2,002	11,081
Total data transferred to/from disk (TB)	5383	74,650	514,754
Avg. MB transferred to/from disk per task second (MB/s)	.787	1.18	1.47
Avg. Mb sent over the network per task second (Mbps)	1.34	1.61	1.44

Table 2: Disk use for all MapReduce jobs run at Google.

	May	Jun	Jul	Aug	Sep	Oct	Nov	Dec	Jan
Input Data (PB) [11]	12.6	22.7	14.3	18.7	22.8	25.3	25.0	18.6	21.5
Intermediate Data (PB) [11]	0.66	1.22	0.67	0.76	0.73	0.86	0.68	0.72	1.99
Compute (years) [11]	49.1	88.0	51.6	60.6	73.0	84.1	88.4	96.2	79.5
Avg. MB read/written per task second	8.99	9.06	9.61	10.58	10.54	10.19	9.46	6.61	10.16
Avg. Mb shuffled per task second	3.41	3.52	3.29	3.18	2.54	2.59	1.95	1.90	6.35

Table 3: Disk use for a cluster with tens of thousands of machines, running Cosmos. Compute (years) describes the sum of runtimes across all tasks [12]. The data includes jobs from two days of each month; see [11] for details.

I/O requirements than the workloads we measure.

3.5 Why isn't disk I/O more important?

We were surprised at the results in §3.3 given the oft-quoted mantra that I/O is often the bottleneck, and also the fact that fundamentally, the computation done in data analytics job is often very simple. For example, queries 1a, 1b, and 1c in the big data benchmark select a filtered subset of a table. Given the simplicity of that computation, we would not have expected the query to be CPU bound. One reason for this result is that today's frameworks often store compressed data (in increasingly sophisticated formats, e.g. Parquet [1]), trading CPU time for I/O time. We found that if we instead ran queries on uncompressed data, most queries became I/O bound. A second reason that CPU time is large is an artifact of the decision to write Spark in Scala, which is based on Java: after being read from disk, data must be deserialized from a byte buffer to a Java object. Figure 7 illustrates the distribution of the total non-idle CPU time used by queries in the big data benchmark under 3 different scenarios: when input data, shuffle data, and output data are compressed and serialized; when input data, shuffle data, and output data are not deserialized but are decompressed; and when input and output data are stored as deserialized, in-memory objects (shuffle data must still be serialized in order to be sent over the network). The CDF illustrates that for some queries, as much as half of the CPU time is spent deserializing and decompressing data. This result is consistent with Figure 9 from the Spark paper, which illustrated that caching deserialized data significantly reduced job completion time relative to caching data that was still serialized.

Spark's relatively high CPU time may also stem from the fact that Spark was written Scala, as opposed to a lower-level language such at C++. For one query that we re-wrote in C++, we found that the CPU time reduced by a factor of more than 2×. Existing work has illustrated that writing analytics in C++ instead can significantly improve performance [22], and the fact that Google's MapReduce is written in C++ is an oft-quoted reason for its superior performance.

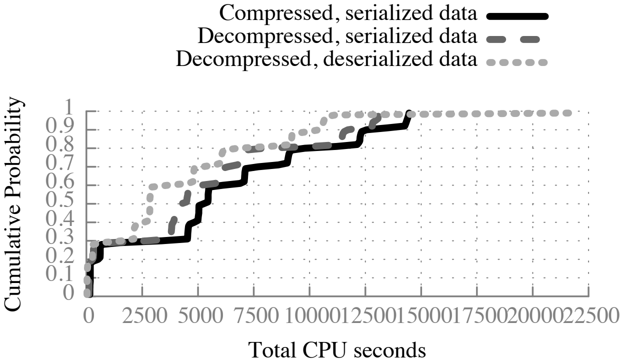


Figure 7: Comparison of total non-idle CPU milliseconds consumed by the big data benchmark workload, with and without compression and serialization.

3.6 Are these results inconsistent with past work?

Prior work has shown significant improvements as a result of storing input data for analytics workloads in memory. For example, Spark [51] was demonstrated to be 20× to 40× faster than Hadoop [51]. A close reading of that paper illustrates that much of that improvement came not from eliminating disk I/O, but from other improvements over Hadoop, including eliminating serialization time.

The PACMan work described improvements in average job completion time of more than a factor of two as a result of caching input data [9], which, similar to Spark, seems to potentially contradict our results. The 2× improve-

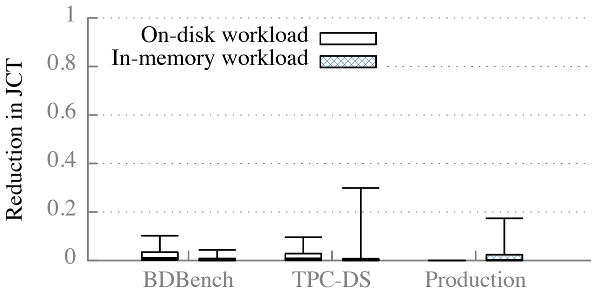


Figure 8: Improvement in job completion time as a result of eliminating all time blocked on network.

ments were shown for two workloads. The first workload was based on the Facebook trace, but because the Facebook trace does not include enough information to replay the exact computation used by the jobs, the authors used a mix of compute-free (jobs that do not perform any computation), sort, and word count jobs [12]. This synthetic workload was generated based on the assumption that analytics workloads are I/O bound, which was the prevailing mentality at the time. Our measurements suggest that jobs are not typically I/O bound, so this workload may not have been representative. The second workload was based on a Bing workload (rewritten to use Hive), where the 2× improvement represented the difference in reading data from a serialized, on-disk format compared to reading de-serialized, in-memory data. As discussed in §3.5, serialization times can be significant, so the 2× improvement likely came as much from eliminating serialization as it did from eliminating disk I/O.

3.7 Summary

We found that job runtime cannot improve by more than 19% as a result of optimizing disk I/O. To shed more light on this measurement,we compared resource utilization while tasks were running, and found that CPU utilization is typically close to 100% whereas median disk utilization is at most 25%. One reason for the relatively high use of CPU by the analytics workloads we studied is deserialization and compression; the shift towards more sophisticated serialization and compression formats has decreased the I/O and increased the CPU requirements of analytics frameworks. Because of high CPU times, optimizing hardware performance by using more disks, using flash storage, or storing serialized in-memory data will yield only modest improvements to job completion time; caching deserialized data has the potential for much larger improvements due to eliminating deserialization time.

Serialization and compression formats will inevitably evolve in the future, rendering the numbers presented in this paper obsolete. The takeaway from our measurements should be that CPU usage is currently much higher than disk use, and that detailed performance instrumentation like our blocked time analysis is critical to navigating the tradeoff between CPU and I/O time going forward.

4 How important is the network?

Researchers have used the argument that data-intensive application performance is closely tied to datacenter network performance to justify a wide variety of network optimizations [6, 14, 17–21, 27, 28, 41, 42, 48, 53]. We therefore expected to find that optimizing the network could yield significant improvements to job completion time.

4.1 How much time is spent blocked on network I/O?

To understand the importance of the network, we first use blocked time analysis to understand the largest possible improvement from optimizing the network. As shown in Figure 8, none of the workloads we studied could improve by a median of more than 2% as a result of optimizing network performance. We did not use especially high bandwidth machines in getting this result: the m2.4xlarge instances we used have a 1Gbps network link.

Our blocked time instrumentation for the network included time to read shuffle data over the network, and for on-disk workloads, the time to write output data to one local machine and two remote machines. Both of these times include disk use as well as network use, because disk and network are interlaced in a manner that makes them difficult to measure separately. As a result, 2% represents an upper bound on the possible improvement from network optimizations.

To shed more light on the network demands of the workloads we ran, Figure 5 plots the network utilization along with CPU and disk utilization. Consistent with the fact that blocked times are very low, median network utilization is lower than median disk utilization and much lower than median CPU utilization for all of the workloads we studied.

4.2 Sanity-checking our results against production traces

We were surprised at the low impact of the network on job completion time, given the large body of work targeted at improving network performance for analytics workloads. Similar to what we did in §3.3 to understand disk performance, we computed the network data sent per non-idle CPU second, to facilitate comparison to large-scale production workloads, as shown in Figure 9. Similar to what we found for disk I/O, the Facebook workload transfers less data over the network per CPU second than the workloads we ran. Thus, we expect that running our blocked time analysis on the Facebook workload would yield smaller potential improvements from network optimizations than for the workloads we ran. Tables 2 and 3 illustrate this metric for the Google and Microsoft traces, using the machine seconds or task seconds

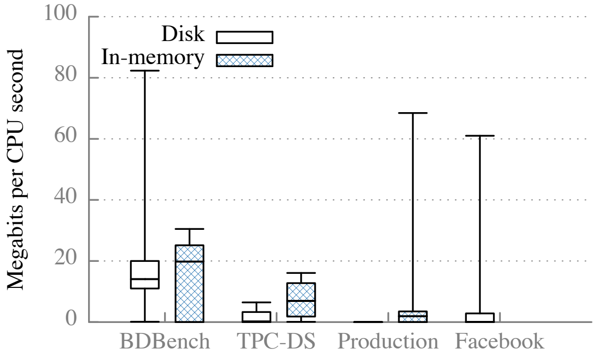


Figure 9: Megabits sent over the network per non-idle CPU second.

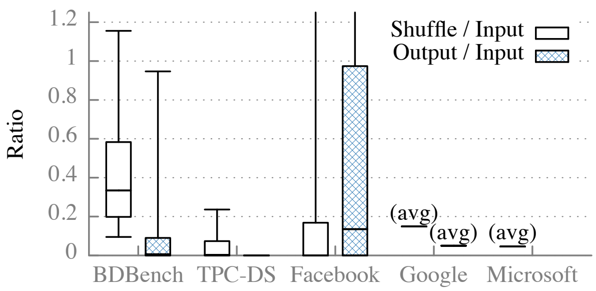


Figure 10: Ratio of shuffle bytes to input bytes and output bytes to input bytes. The median ratio of shuffled data to input data is less than 0.35 for all workloads, and the median ratio of output data to input data is less than 0.2 for all workloads.

in place of CPU milliseconds as with the disk results.[7] For Google, the average megabits sent over the network per machine second ranges from 1.34 to 1.61; Microsoft network use is higher (1.90-6.35 megabits are shuffled per task second) but still far below the network use seen in our workload.Thus, this sanity check does not lead us to believe that production workloads have dramatically different network requirements than the workloads we measure.

4.3 Why isn't the network more important?

One reason that network performance is relatively unimportant is that the amount of data sent over the network is often much less than the data transferred to disk, because analytics queries often shuffle and output much less data than they read. Figure 10 plots the ratio of shuffle data to input data and the ratio of output data to input data across our benchmarks, the Facebook trace (the production workload did not have sufficient instrumentation to capture these metrics), and for the Microsoft and Google aggregate data (averaged over all of the samples). Across all workloads, the amount of data shuffled is less than the amount of input data, by as much as a factor of 5-10, which is intuitive considering that data analysis often centers around aggregating data to derive a business conclusion. In evaluating the efficacy of optimizing shuffle performance, many papers have used workloads with ratios of shuffled data to input data of 1 or more, which these results demonstrate is not widely representative.

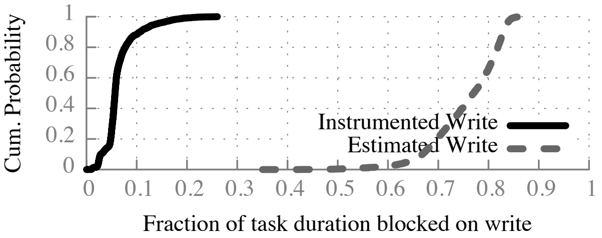


Figure 11: Cumulative distribution across tasks (weighted by task duration) of the fraction of task time spent writing output data, measured using our fine grained instrumentation and estimated using the technique employed by prior work [17]. The previously used metric far overestimates time spent writing output data.

4.4 Are these results inconsistent with past work?

Past work has reported high effects of the network on job completion time because the effect of the network has been estimated based on Hadoop traces rather than precisely measured. Many existing traces do not include sufficient metrics to understand the impact of the network. For example, Sinbad [17] asserted that writing output data can take a significant amount of time for analytics jobs, but used a heuristic to understand time to write output data: it defined the write phase of a task as the time from when the shuffle completes until the completion time of the task. This metric is an estimate, which was necessary because Hadoop does not log time blocked on output writes separately from time spent in computation. We instrumented Hadoop to log time spent writing output data and ran the big data benchmark (using Hive to convert SQL queries to Map Reduce jobs) and compared the result from the detailed instrumentation to the estimation previously used. Unfortunately, as shown in Figure 11, the previously used metric significantly overestimates time spent writing output data, meaning that the importance of the network was significantly overestimated.

A second problem with past estimations of the importance of the network is that they have conflated inefficiencies in Hadoop with network performance problems. One commonly cited work quotes the percent of job time spent in shuffle, measured using a Facebook Hadoop workload [19]. We repeated this measurement using the Facebook trace, shown in Table 4. Previous measurements looked at the fraction of jobs that spent a certain percent of time in shuffle (i.e., the first two lines of Table 4); by this metric, 16% of jobs spent more than 75% of time shuffling data. We dug into this data and found that a typical job that spends more than 75% of time in its shuf-

[7] The Microsoft data does not include output size, so network data only includes shuffle data.

Shuffle Dur.	<25%	25-49%	50-74%	>=75%
% of Jobs	46%	20%	18%	16%
% of time	91%	7%	2%	1%
% of bytes	83%	16%	1%	0.3%

Table 4: The percent of jobs, task time, and bytes in jobs that spent different fractions of time in shuffle for the Facebook workload.

fle phase takes tens of seconds to shuffle kilobytes of data, suggesting that framework overhead and not network performance is the bottleneck. We also found that only 1% of all jobs spend less than 4 seconds shuffling data, which suggests that the Hadoop shuffle includes fixed overheads to, for example, communicate with the master to determine where shuffle data is located. For such tasks, shuffle is likely bottlenecked on inefficiencies and fixed overheads in Hadoop, rather than by network bottlenecks.

Table 4 includes data not reported in prior work: for each bucket, we compute not only the percent of jobs that fall into that bucket, but also the percent of total time and percent of total bytes represented by jobs in that category. While 16% of jobs spend more than 75% of the time in shuffle, these jobs represent only 1% of the total bytes sent across the network, and 0.3% of the total time taken by all jobs. This further suggests that the jobs reported as spending a large fraction of time in shuffle are small jobs for which the shuffle time is dominated by framework overhead rather than by network performance.

4.5 Summary

We found that, for the three workloads we studied, network optimizations can only improve job completion time by a median of at most 2%. One reason network performance has little effect on job completion time is that the data transferred over the network is a subset of data transferred to disk, so jobs bottleneck on the disk before bottlenecking on the network. We found this to be true in a cluster with 1Gbps network links; in clusters with 10Gbps or 100Gbps networks, network performance will be even less important.

Past work has found much larger improvements from optimizing network performance for two reasons. First, some past work has focused only on workloads where shuffle data is equal to the amount of input data, which we demonstrated is not representative of typical workloads. Second, some past work relied on estimation to understand trace data, which led to misleading conclusions about the importance of the network.

5 The Role of Stragglers

A straggler is a task that takes much longer to complete than other tasks in the stage. Because the stage cannot complete until all of its tasks have completed, straggler tasks can significantly delay the completion time of the stage and, subsequently, the completion time of the higher-level query. Past work has demonstrated that stragglers can be a significant bottleneck; for example, stragglers were reported to delay average job completion time by 47% in Hadoop clusters and by 29% in Dryad clusters [8]. Existing work has characterized some stragglers as being caused by data skew, where a task reads more input data than other tasks in a job, but otherwise does not characterize what causes some tasks to take longer than others [11, 26, 29]. This inability to explain the cause of stragglers has typically led to mitigation strategies that replicate tasks, rather than strategies that attempt to understand and eliminate the root cause of long task runtimes [8, 10, 39, 52].

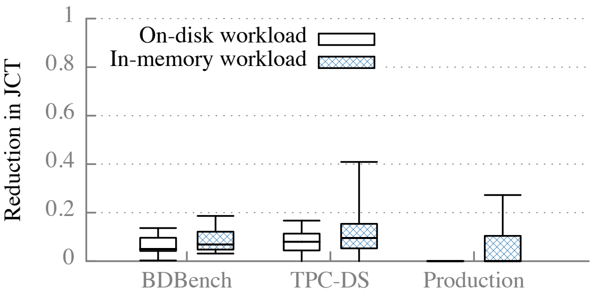


Figure 12: Potential improvement in job completion time as a result of eliminating all stragglers. Results are shown for the on-disk and in-memory versions of each benchmark workload.

5.1 How much do stragglers affect job completion time?

To understand the impact of stragglers, we adopt the approach from [8] and focus on a task's inverse progress rate: the time taken for a task divided by amount of input data read. Consistent with that work, we define the potential improvement from eliminating stragglers as the reduction in job completion time as a result of replacing the progress rate of every task that is slower than the median progress rate with the median progress rate. We use the methodology described in §2.3.2 to compute the new job completion time given the new task completion times. Figure 12 illustrates the improvement in job completion time as a result of eliminating stragglers in this fashion; the median improvement from eliminating stragglers is 5-10% for the big data benchmark and TPC-DS workloads, and lower for the production workloads, which had fewer stragglers.

5.2 Are these results inconsistent with prior work?

Some prior work has reported the effect of stragglers to be similar to what we found in our study. For example, based on a Facebook trace, Mantri described a median improvement of 15% as a result of eliminating all stragglers. Mantri had a larger overall improvements when deployed in production that stemmed not only from eliminating stragglers, but also from eliminating costly

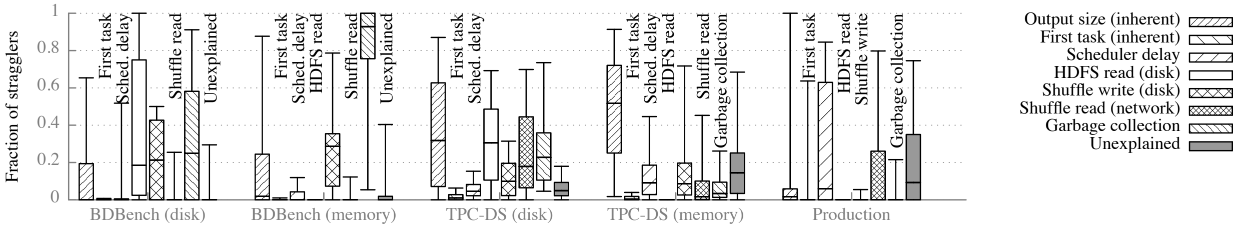


Figure 13: Our improved instrumentation allowed us to explain the causes behind most of the stragglers in the workloads we instrumented. This plot shows the distribution across all queries of the fraction of the query's stragglers that can be attributed to a particular cause.

recomputations [11].

Other work has reported larger potential improvements from eliminating stragglers: Dolly, for example, uses the same metric we used to understand the impact of stragglers, and found that eliminating stragglers could reduce job completion time by 47% for a Facebook trace and 29% for a Bing trace [8]. This difference may be due to the larger cluster size or greater heterogeneity in the studied traces. However, the difference can also be partially attributed to framework differences. For example, Spark has much lower task launch overhead than Hadoop, so Spark often breaks jobs into many more tasks, which can reduce the impact of stragglers [39]. Stragglers would have had a much larger impact for the workloads we ran if all of the tasks in each job had run as a single wave; in this case, the median improvement from eliminating stragglers would have been 23-41%. We also found that stragglers have become less important as Spark has matured. When running the same benchmark queries with an older version of Spark, many stragglers were caused by poor disk performance when writing shuffle data. Once this problem was fixed, the importance of stragglers decreased. These improvements to Spark may make stragglers less of an issue than previously observed, even on large-scale clusters.

5.3 Why do stragglers occur?

Prior work investigating straggler causes has largely relied on traces with coarse grained instrumentation; as a result, this work has been able to attribute stragglers to data skew and high resource utilization [11, 49], but otherwise has been unable to explain why stragglers occur. Our instrumentation allows us to describe the cause of more than 60% of stragglers in 75% of the queries we ran.

In examining the cause of stragglers, we follow previous work and define a straggler as a task with inverse progress rate greater than 1.5× the median inverse progress rate for the stage. Unlike the previous subsection, we do not look at all tasks that take longer than the median progress rate, in order to focus on situations where there was a significant anomaly in a task's execution.

Many stragglers can be explained by the fact that the straggler task spends an unusually long amount of time in a particular part of task execution. We characterize a straggler as caused by *X* if it would not have been considered a straggler had *X* taken zero time for all of the tasks in the stage. We use this methodology to attribute stragglers to scheduler delay (time taken by the scheduler to ship the task to a worker and to process the task completion message), HDFS disk read time, shuffle write time, shuffle read time, and Java's garbage collection (which can be measured using Java's `GarbageCollectorMXBean` interface).

We attribute stragglers to two additional causes that require different methodologies. First, we attribute stragglers to output skew by computing the progress rate based on the amount of output data processed by the task instead of the amount of input data, and consider a straggler caused by output skew if the task is a straggler based on input progress rate but not based on output progress rate. Second, we find that some stragglers can be explained by the fact that they were among the first tasks in a stage to run on a particular machine. This effect may be caused by Java's just-in-time compilation: Java runtimes (e.g., the HotSpot JVM [32]) optimize code that has been executed more than a threshold number of times. We consider stragglers to be caused by the fact that they were a "first task" if they began executing before any other tasks in the stage completed on the same machine, and if they are no longer considered stragglers if compared to other "first tasks."

For each of these causes, Figure 13 plots the fraction of stragglers in each query explained by that cause. The distribution arises from differences in straggler causes across queries; for example, for the on-disk big data benchmark, in some jobs, all stragglers are explained by the time to read data from HDFS, whereas in other jobs, most stragglers can be attributed to garbage collection.

The graph does not point to any one dominant of stragglers, but rather illustrates that straggler causes vary across workloads and even within queries for a particular workload. However, common patterns are that garbage collection can cause most of the stragglers for some queries, and many stragglers can be attributed to long times spent reading to or writing from disk (this is not inconsistent with our earlier results showing a 19% median improvement from eliminating disk: the fact that some straggler tasks are caused by long times spent

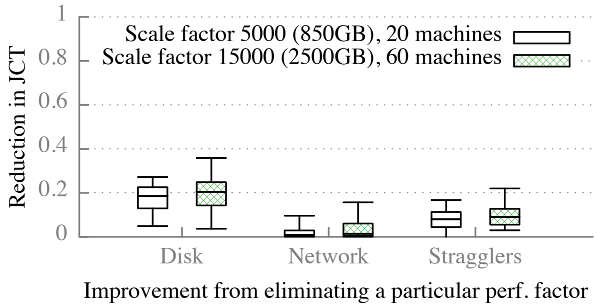


Figure 14: Improvement in job completion time as a result of eliminating disk I/O, eliminating network I/O, and eliminating stragglers, each shown for two different trials of the on-disk TPC-DS workload using different scale factors and cluster sizes.

blocked on disk does not necessarily imply that overall, eliminating time blocked on disk would yield a large improvement in job completion time). Another takeaway is that many stragglers are caused by inherent factors like output size and running before code has been jitted, so cannot be alleviated by straggler mitigation techniques.

5.4 Improving performance by understanding stragglers

Understanding the root cause behind stragglers provides ways to improve performance by mitigating the underlying cause. In our early experiments, investigating straggler causes led us to find that the default file system on the EC2 instances we used, ext3, performs poorly for workloads with large numbers of parallel reads and writes, leading to stragglers. By changing the filesystem to ext4, we fixed stragglers and reduced median task time, reducing query runtime for queries in the big data benchmark by $17-58\%$. Many of the other stragglers we observed could potentially be reduced by targeting the underlying cause, for example by allocating fewer objects (to target GC stragglers) or consolidating map output data into fewer files (to target shuffle write stragglers). Understanding these causes allows for going beyond duplicating tasks to mitigate stragglers.

6 How Does Scale Affect Results?

The results in this paper have focused on one cluster size for each of the benchmarks run. To understand how our results might change in a much larger cluster, we ran the TPC-DS on-disk workload on a cluster with three times as many machines and using three times more input data. Figure 14 compares our key results on the larger cluster to the results from the 20-machine cluster described in the remainder of this paper, and illustrates that the potential improvements from eliminating disk I/O, eliminating network I/O, and eliminating stragglers on the larger cluster is comparable to the corresponding improvements on the smaller cluster.

7 Conclusion

This paper undertook a detailed performance study of three workloads, and found that for those workloads, jobs are often bottlenecked on CPU and not I/O, network performance has little impact on job completion time, and many straggler causes can be identified and fixed. These findings should not be taken as the last word on performance of analytics frameworks: our study focuses on a small set of workloads, and represents only one snapshot in time. As data-analytics frameworks evolve, we expect bottlenecks to evolve as well. As a result, the takeaway from this work should be the importance of instrumenting systems for blocked time analysis, so that researchers and practitioners alike can understand how best to focus performance improvements. Looking forward, we argue that systems should be built with performance understandability as a first-class concern. Obscuring performance factors sometimes seems like a necessary cost of implementing new and more complex optimizations, but inevitably makes understanding how to optimize performance in the future much more difficult.

8 Acknowledgments

We are thankful to many Databricks employees who made the inclusion of production data possible: Aaron Davidson, Ali Ghodsi, Lucian Popa, Ahir Reddy, Ion Stoica, Patrick Wendell, Reynold Xin, and especially Andy Konwinski. We are also thankful to Michael Armbrust, for helping to configure and debug SparkSQL; to Yanpei Chen, Radhika Mittal, Amy Ousterhout, Aurojit Panda, and Shivaram Venkataraman, for providing helpful feedback on drafts of this paper; and to Mosharaf Chowdhury, for helping to analyze Facebook trace data. Finally, we thank our shepherd, Venugopalan Ramasubramanian, for helping to shape the final version of the paper.

This research is supported in part by a Hertz Foundation Fellowship, NSF CISE Expeditions Award CCF-1139158, LBNL Award 7076018, and DARPA XData Award FA8750-12-2-0331, and gifts from Amazon Web Services, Google, SAP, The Thomas and Stacey Siebel Foundation, Adatao, Adobe, Apple, Inc., Blue Goji, Bosch, C3Energy, Cisco, Cray, Cloudera, EMC, Ericsson, Facebook, Guavus, Huawei, Informatica, Intel, Microsoft, NetApp, Pivotal, Samsung, Splunk, Virdata and VMware.

References

[1] Apache Parquet. `http://parquet.incubator.apache.org/`.

[2] Common Crawl. `http://commoncrawl.org/`.

[3] Databricks. `http://databricks.com/`.

[4] Spark SQL. `https://spark.apache.org/sql/`.

[5] M. K. Aguilera, J. C. Mogul, J. L. Wiener, P. Reynolds, and A. Muthitacharoen. Performance Debugging for Distributed Systems of Black Boxes. In Proc. SOSP, 2003.

[6] M. Al-fares, S. Radhakrishnan, B. Raghavan, N. Huang, and A. Vahdat. Hedera: Dynamic Flow Scheduling for Data Center Networks. In Proc. NSDI, 2010.

[7] G. Ananthanarayanan, S. Agarwal, S. Kandula, A. Greenberg, I. Stoica, D. Harlan, and E. Harris. Scarlett: Coping with Skewed Content Popularity in MapReduce Clusters. In Proc. EuroSys, 2011.

[8] G. Ananthanarayanan, A. Ghodsi, S. Shenker, and I. Stoica. Effective Straggler Mitigation: Attack of the Clones. In Proc. NSDI, 2013.

[9] G. Ananthanarayanan, A. Ghodsi, A. Wang, D. Borthakur, S. Kandula, S. Shenker, and I. Stoica. PACMan: Coordinated Memory Caching for Parallel Jobs. In Proc. NSDI, 2012.

[10] G. Ananthanarayanan, M. C.-C. Hung, X. Ren, I. Stoica, A. Wierman, and M. Yu. GRASS: Trimming Stragglers in Approximation Analytics. In Proc. NSDI, 2014.

[11] G. Ananthanarayanan, S. Kandula, A. Greenberg, I. Stoica, Y. Lu, B. Saha, and E. Harris. Reining in the Outliers in Map-Reduce Clusters using Mantri. In Proc. OSDI, 2010.

[12] G. Ananthanarayayan. Personal Communication, February 2015.

[13] Apache Software Foundation. Apache Hadoop. `http://hadoop.apache.org/`.

[14] H. Ballani, P. Costa, T. Karagiannis, and A. Rowstron. Towards Predictable Datacenter Networks. In Proc. SIGCOMM, 2011.

[15] P. Barham, A. Donnelly, R. Isaacs, and R. Mortier. Using magpie for request extraction and workload modelling. In Proc. SOSP, 2004.

[16] D. Borthakur. Facebook has the world's largest Hadoop cluster! `http://hadoopblog.blogspot.com/2010/05/facebook-has-worlds-largest-hadoop.html`, May 2010.

[17] M. Chowdhury, S. Kandula, and I. Stoica. Leveraging Endpoint Flexibility in Data-intensive Clusters. In Proc. SIGCOMM, 2013.

[18] M. Chowdhury and I. Stoica. Coflow: A Networking Abstraction for Cluster Applications. In Proc. HotNets, 2012.

[19] M. Chowdhury, M. Zaharia, J. Ma, M. I. Jordan, and I. Stoica. Managing Data Transfers in Computer Clusters with Orchestra. In Proc. SIGCOMM, 2011.

[20] M. Chowdhury, Y. Zhong, and I. Stoica. Efficient Coflow Scheduling with Varys. In Proc. SIGCOMM, 2014.

[21] P. Costa, A. Donnelly, A. Rowstron, and G. O'Shea. Camdoop: Exploiting In-network Aggregation for Big Data Applications. In Proc. NSDI, 2012.

[22] A. Crotty, A. Galakatos, K. Dursun, T. Kraska, U. Çetintemel, and S. B. Zdonik. Tupleware: Redefining modern analytics. CoRR, 2014.

[23] J. Dean. Personal Communication, February 2015.

[24] J. Dean and S. Ghemawat. MapReduce: Simplified Data Processing on Large Clusters. CACM, 51(1):107–113, Jan. 2008.

[25] J. Erickson, M. Kornacker, and D. Kumar. New SQL Choices in the Apache Hadoop Ecosystem: Why Impala Continues to Lead. `http://goo.gl/evDBfy`, 2014.

[26] B. Gufler, N. Augsten, A. Reiser, and A. Kemper. Load Balancing in MapReduce Based on Scalable Cardinality Estimates. In Proc. ICDE, pages 522–533, 2012.

[27] Z. Guo, X. Fan, R. Chen, J. Zhang, H. Zhou, S. McDirmid, C. Liu, W. Lin, J. Zhou, and L. Zhou. Spotting Code Optimizations in Data-Parallel Pipelines through PeriSCOPE. In Proc. OSDI, 2012.

[28] V. Jeyakumar, M. Alizadeh, D. Mazieres, B. Prabhakar, C. Kim, and A. Greenberg. EyeQ: Practical Network Performance Isolation at the Edge. In Proc. NSDI, 2013.

[29] Y. Kwon, M. Balazinska, B. Howe, and J. Rolia. SkewTune: Mitigating Skew in MapReduce Applications. In Proc. SIGMOD, pages 25–36, 2012.

[30] H. Li, A. Ghodsi, M. Zaharia, S. Shenker, and I. Stoica. Reliable, Memory Speed Storage for Cluster Computing Frameworks. In Proc. SoCC, 2014.

[31] K. O'Dell. How-to: Select the Right Hardware for Your New Hadoop Cluster. `http://goo.gl/INds4t`, August 2013.

[32] Oracle. The Java HotSpot Performance Engine Architecture. http://www.oracle.com/technetwork/java/whitepaper-135217.html.

[33] K. Ousterhout. Display filesystem read statistics with each task. https://issues.apache.org/jira/browse/SPARK-1683.

[34] K. Ousterhout. Shuffle read bytes are reported incorrectly for stages with multiple shuffle dependencies. https://issues.apache.org/jira/browse/SPARK-2571.

[35] K. Ousterhout. Shuffle write time does not include time to open shuffle files. https://issues.apache.org/jira/browse/SPARK-3570.

[36] K. Ousterhout. Shuffle write time is incorrect for sort-based shuffle. https://issues.apache.org/jira/browse/SPARK-5762.

[37] K. Ousterhout. Spark big data benchmark and TPC-DS workload traces. http://eecs.berkeley.edu/~keo/traces.

[38] K. Ousterhout. Time to cleanup spilled shuffle files not included in shuffle write time. https://issues.apache.org/jira/browse/SPARK-5845.

[39] K. Ousterhout, A. Panda, J. Rosen, S. Venkataraman, R. Xin, S. Ratnasamy, S. Shenker, and I. Stoica. The Case for Tiny Tasks in Compute Clusters. In Proc. HotOS, 2013.

[40] A. Pavlo, E. Paulson, A. Rasin, D. J. Abadi, D. J. DeWitt, S. Madden, and M. Stonebraker. A Comparison of Approaches to Large-scale Data Analysis. In Proc. SIGMOD, 2009.

[41] L. Popa, G. Kumar, M. Chowdhury, A. Krishnamurthy, S. Ratnasamy, and I. Stoica. FairCloud: Sharing The Network in Cloud Computing. In Proc. SIGCOMM, 2012.

[42] P. Prakash, A. Dixit, Y. C. Hu, and R. Kompella. The TCP Outcast Problem: Exposing Unfairness in Data Center Networks. In Proc. NSDI, 2012.

[43] A. Rasmussen, V. T. Lam, M. Conley, G. Porter, R. Kapoor, and A. Vahdat. Themis: An I/O-efficient MapReduce. In Proc. SoCC, 2012.

[44] K. Sakellis. Track local bytes read for shuffles - update UI. https://issues.apache.org/jira/browse/SPARK-5645.

[45] Transaction Processing Performance Council (TPC). TPC Benchmark DS Standard Specification. http://www.tpc.org/tpcds/spec/tpcds_1.1.0.pdf, 2012.

[46] UC Berkeley AmpLab. Big Data Benchmark. https://amplab.cs.berkeley.edu/benchmark/, February 2014.

[47] A. Wang and C. McCabe. In-memory Caching in HDFS: Lower Latency, Same Great Taste. In Presented at Hadoop Summit, 2014.

[48] D. Xie, N. Ding, Y. C. Hu, and R. Kompella. The Only Constant is Change: Incorporating Time-Varying Network Reservations in Data Centers. In Proc. SIGCOMM, 2012.

[49] N. J. Yadwadkar, G. Ananthanarayanan, and R. Katz. Wrangler: Predictable and Faster Jobs Using Fewer Resources. In Proc. SoCC, 2014.

[50] L. Yi, K. Wei, S. Huang, and J. Dai. Hadoop Benchmark Suite (HiBench). https://github.com/intel-hadoop/HiBench, 2012.

[51] M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauley, M. J. Franklin, S. Shenker, and I. Stoica. Resilient Distributed Datasets: A Fault-Tolerant Abstraction for In-Memory Cluster Computing. In Proc. NSDI, 2012.

[52] M. Zaharia, A. Konwinski, A. D. Joseph, R. Katz, and I. Stoica. Improving MapReduce Performance in Heterogeneous Environments. In Proc. OSDI, 2008.

[53] J. Zhang, H. Zhou, R. Chen, X. Fan, Z. Guo, H. Lin, J. Y. Li, W. Lin, J. Zhou, and L. Zhou. Optimizing Data Shuffling in Data-Parallel Computation by Understanding User-Defined Functions. In Proc. NSDI, 2012.

CellIQ: Real-Time Cellular Network Analytics at Scale

Anand Padmanabha Iyer[★], Li Erran Li[†], Ion Stoica[★]

[★] University of California, Berkeley [†] Bell Labs, Alcatel-Lucent

Abstract

We present CellIQ, a real-time cellular network analytics system that supports rich and sophisticated analysis tasks. CellIQ is motivated by the lack of support for real-time analytics or advanced tasks such as spatio-temporal traffic hotspots and handoff sequences with performance problems in state-of-the-art systems, and the interest in such tasks by network operators. CellIQ represents cellular network data as a stream of domain specific graphs, each from a batch of data. Leveraging domain specific characteristics—the spatial and temporal locality of cellular network data—CellIQ presents a number of optimizations including geo-partitioning of input data, radius-based message broadcast, and incremental graph updates to support efficient analysis. Using data from a live cellular network and representative analytic tasks, we demonstrate that CellIQ enables fast and efficient cellular network analytics—compared to an implementation without cellular specific operators, CellIQ is 2× to 5× faster.

1 Introduction

Cellular networks have become an integral part of our digital life in an increasingly mobile connected world driven by the wide adoption of smartphones and tablets. These networks must be designed, operated and maintained efficiently in order to ensure satisfactory end-user experience. To achieve this goal, cellular network operators collect unprecedented volume of information about the network and user traffic. Analysis of this data can provide crucial insights in a number of tasks ranging from network planning (e.g., deployment of new base stations) to network operation (e.g., improving utilization of limited radio resources by interference coordination). The analysis is not limited to network operations—for instance, it can also help cities plan smarter road networks, businesses reach more potential customers, and health officials track diseases [29]. Thus, timely and efficient analysis of cellular network data can be beneficial in a variety of scenarios.

Current state-of-the-art cellular analytics systems continuously collect per connection information such as radio resource usage, associated base stations and handoffs at network elements such as the Mobility Management Entity (MME) and probes deployed in strategic locations. The collected information is then backhauled to centralized servers in batches and ingested into the analysis engine [3, 5, 15, 35]. Such analytics systems are either based on streaming database technology [15] or Hadoop batch processing [5, 35]. Thousands of predefined reports are generated from the data periodically and made available in a dashboard for the experts to view. While these reports provide useful information, we have learned from network operators that they can benefit from *timely* and more *sophisticated* analyses. For example, advanced analytics such as detecting and monitoring spatio-temporal hotspots and tracking popular handoff sequences with abnormal failure rate would enable quick resolution of performance problems.

In this paper, we propose CellIQ, a system for cellular network analytics that builds on top of existing big data cluster computing frameworks. By leveraging domain specific knowledge, CellIQ enables fast and efficient cellular network analysis at scale. The key insight in CellIQ is the observation that cellular network data is naturally represented as a time-evolving graph. In this graph, nodes are network entities such as base stations and User Equipments (UE). Edges represent adjacency of base stations or connections between base stations and UEs.

Stream processing and graph processing has been topics of tremendous interest recently, and hence a large number of proposals exist in both areas. Existing streaming systems such as TimeStream [31] and Spark Streaming [37] do not support streaming graph processing. On the other hand, existing graph parallel systems such as GraphLab [25], PowerGraph [19], GraphX [18] and GraphLINQ [28] are not optimized for operations spanning multiple graphs such as persistent connected components over sliding windows. There are a couple of noticeable exceptions: Kineograph [11] and Chronos [21] focus on constructing incremental snapshots of evolving graphs and optimizing data layout and job scheduling. Differential Dataflow [27] supports incremental computation of algorithms on evolving graphs in the Naiad [30] framework. These systems do not present specific optimizations for cellular network analytics.

In contrast, CellIQ is optimized for cellular network analytics. It leverages domain specific characteristics of cellular networks—its *spatial and temporal locality*—to achieve efficient analysis. CellIQ encodes network specific properties in a time-evolving graph. Connection records per time window are edge properties between UEs and base stations. Aggregate statistics per time window such as radio resource allocation, modulation and

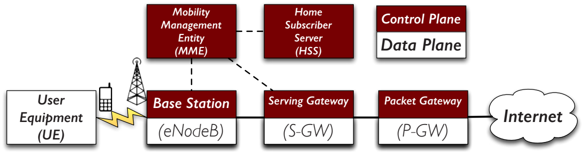


Figure 1: LTE network architecture (description in Table 1).

traffic volume are node properties. It then optimizes the data layout with the use of space-filling curve based geo-partitioning and edge indexing mechanisms. Window operations are efficiently implemented using differential and incremental graph update techniques. To avoid hop-by-hop message propagation, CellIQ enables nodes to broadcast messages to all nodes within a radius. For spatial operations, we further support efficient aggregations.

Using real cellular network data and representative analytics tasks such as spatial and temporal traffic hotspots and popular handoffs, we demonstrate that CellIQ enables real time cellular network analytics. The performance gain can be significant when compared with solutions without cellular specific optimizations.

This paper makes the following contributions:

- We have designed and developed CellIQ, which to the best of our knowledge is the first real-time cellular network analytics system that is capable of running sophisticated tasks including detection and tracking of spatial and temporal traffic hotspots, and popular handoff sequences with abnormal failures.
- We systematically take advantage of the domain specific characteristics of cellular networks, its spatial and temporal locality, to optimize the performance of CellIQ. We succinctly represent a batch of cellular network data as a spatial graph, and continuously arriving data as a stream of spatial graphs. We carefully place data using a geo-partitioning technique that avoids expensive data movements. We propose differential and incremental graph updates for efficient window operations and radius based message broadcast for quicker spatial analysis.
- We evaluate our system using real cellular network data consisting of several thousand base stations and millions of users. Our results show that CellIQ outperforms implementations without cellular specific optimizations vastly.

2 Background on LTE Networks

In this section, we briefly review the LTE network architecture and its data collection mechanism to familiarize the reader with the basic entities in the network and the characteristics of the data available for analysis. We also discuss how existing state-of-the-art analytic systems utilize such collected data.

2.1 LTE Network Architecture

LTE networks enable User Equipments (UEs) such as smartphones to access the Internet. The LTE network architecture is shown in Figure 1, which consists of several network entities (a description is given in Table 1). When a UE is in idle mode, it does not have an active connection to the network. To communicate with the Internet, a UE requests the network to establish a communication channel between itself and the Packet Data Network Gateway (P-GW). This involves message exchanges between the UE and the Mobility Management Entity (MME). The MME may contact the Home Subscriber Server (HSS) to obtain UE capability and credentials. To enable the communication between the UE and MME, a radio connection called radio bearer between the UE and the base station is established. GPRS Tunneling Protocol (GTP) tunnels are established between the base station and the Serving Gateway (S-GW), and between the S-GW and the P-GW through message exchanges involving these entities and the MME. The radio bearer and the two GTP tunnels make up the the communication channel between the UE and the P-GW called Evolved Packet System (EPS) bearer (or simply bearer in short).

When an active UE moves across a base station boundary, its connections will be handed off to the new base station. There are several different types of handoffs: handoffs that require the bearer to be handled by a new S-GW, a new MME, handoffs that require the change of radio frequency or radio technology (e.g. from LTE to 3G). Some of these procedures are very involved. For an active UE, the network knows its current associated base station. For an idle UE, the network knows its current tracking area. A tracking area is a set of base stations that are geographically nearby.

S-GWs are mainly used as mobility anchors to provide seamless mobility. P-GW centralizes most network functions like content filter, firewalls, lawful intercepts, etc. P-GWs sit at the boundary of the cellular networks and the Internet. A typical LTE network can cover a very large geographic area (even as large as a country) and can have a pool of MMEs, S-GWs and P-GWs for reliability and load balancing purposes.

2.2 Data Collection and Analysis

Cellular network operators collect a wide variety of data from their network, a few of which are discussed below:

Bearer and Signaling Records: A UE communicates with the network by establishing one or more bearers. Each bearer may have a different QoS profile or connect to a different IP network. Multiple TCP connections can be carried in one bearer. LTE networks keep track of a rich set of bearer statistics such as (1) traffic volume, frame loss rate in the data link layer, (2) physical radio resources allocated, radio channel quality, modulation and

LTE Architecture Entities	
Name	*Description*
UE	User Equipment: Any device that accesses the network such as smartphones and tablets.
eNodeB	Enhanced Node B: The base station through which UEs access the network.
MME	Mobility Management Entity: Provides roaming and handoff support, UE authentication and paging.
HSS	Home Subscriber Server: Is a central database that contains user and subscription-related information.
S-GW	Serving Gateway: Acts as a mobility anchor.
P-GW	Packet Data Network Gateway: Allocates IP address and centralizes most network functions such as content filter, policy enforcement, lawful intercepts and charging support.

Table 1: Key entities in the LTE network architecture.

coding rate in the physical layer, (3) bearer setup delay, failure reason, (4) associated base station, S-GW, P-GW, MME, and (5) bearer start and end time. LTE networks also collect data on many signaling procedures such as handoff, paging (waking up a UE to receive incoming traffic), attach request. The collection of these data occur at MMEs and base stations, which organize them into records. Each record can have *several hundred* fields. As indicated earlier, LTE networks may have a pool of MMEs, S-GWs and P-GWs. Since a base station can communicate with multiple MMEs, bearer level records need to be merged across MMEs.

TCP Flow Records: Probes can be strategically deployed in the network, e.g., between S-GWs and P-GWs. The purpose of these probes is to collect TCP flow records. The collected flows can then be associated with their corresponding bearer records.

Network Element Records: Network elements such as base stations and MMEs have operational statistics such as aggregate downlink frame transmitted per time window and number of bearers failed per time window. These records are also collected and are normally used for network monitoring purposes.

While collecting data packets continuously is infeasible due to its prohibitive space and resource overhead[1], most of the data mentioned above can be collected without noticeable overhead in operational LTE networks.

Existing state-of-the-art cellular analytics systems are deployed in operator owned data centers. The records that are collected at the network entities are accumulated over short time intervals (e.g., per minute) to be sent to the data center. Although the data is available at minute granularity, existing analysis frameworks do not utilize them as soon as they arrive. Instead, the data is accumulated and used to generate several thousands of pre-defined reports (most of which are aggregate statistics) periodically (typically once a day). The generated reports are displayed in a dashboard where domain experts can peruse them.

3 Motivation and Overview

Current cellular network analytics systems provide invaluable insights to the network operator. The reports they generate are immensely useful for the operator to understand the behavior of their network. However, in the present form, the analysis supported by these systems are rudimentary at best. Most, if not all, of the generated reports are simple aggregate statistics such as downlink or uplink volume per network entity. We have learned from network operators that cellular networks could benefit tremendously from sophisticated analytics. For instance, operators are interested in learning if some regions of the network are hotspots, and if they are, whether they persist. Since such hotspots may indicate insufficient network resources, they are useful for dynamic load balancing or network planning. Similarly, it is important to detect and mitigate abnormal failures to provide satisfactory end-user experience. Thus, there is a need for cellular analytic systems to be both *timely* and *sophisticated*.

We now outline the challenges in developing such a system and make a case for using cellular specific optimizations to achieve the desired goals. Then we present our solution briefly.

3.1 Requirements and Challenges

An operational LTE network serving a large region can have thousands of base stations and millions of users. To keep up with the demand, operators are continuously adding capacity by deploying new base stations. The number of other network elements, such as MMEs, S-GWs, and P-GWs are also on the rise. A typical LTE network can generate several terabytes of monitoring data per minute. The volume of monitoring data has been growing as both mobile data-plane traffic and signaling traffic (known as the signaling storm problem due to chatty applications) continue to grow exponentially. Due to the interplay between the network elements, LTE network data has to be analyzed as a whole.

Cellular network operators need to perform a *myriad* of analytics tasks in real time. For example, Motive [4] provides over 7000 offline network analytics reports. Performing these and more advanced analyses real-time means that each computationally intensive task must be executed efficiently to avoid impacting the overall system. To illustrate the challenges involved in performing these analysis, we present three broad tasks that are of interest to network operators:

Continuous monitoring of connections and entities: Operators must continuously monitor millions of UEs, their connections and network elements. Fine-grained location and time-dependent thresholds are needed to prevent unacceptable error alarms.

Real time detection of spatial and temporal patterns: Cellular networks exhibit rich dynamics in both

[1] An operator may choose to enable packet collection for a short duration for troubleshooting purposes, but such cases are typically rare.

temporal and spatial domains. User perceived performance tends to vary over time and location due to changes in the subscribers' activity. Hence, operators need to detect and track spatial and temporal patterns. Examples of such patterns include (1) persistent spatial hotspots in terms of abnormally high signaling traffic, or high frame loss rate, and (2) impending flash crowd events that draws a large number of users to the same location.

Real time troubleshooting to identify root causes: Operators need to perform sophisticated on demand analytics tasks to understand the root cause of performance and security problems. Similar to wired networks, cellular network operators need to detect, locate and troubleshoot performance and security problems, e.g., via expert rule-based inference [24], machine-learning techniques [1, 13], or inference of dependency among network elements, entities and events [7, 22]. Performing these tasks in real-time can be very challenging.

3.2 Need for Cellular Specific Optimizations

Having discussed the various requirements and challenges, we now turn our attention towards how a data processing system may accommodate such analysis tasks. To do so, we contacted network administrators and discussed a few representative analysis tasks of interest. We then implemented these tasks in an existing graph-parallel analysis framework. During this exercise, we realized that most, if not all, of the analysis tasks can be expressed by three operations: (i) sliding window operation, (ii) time window operation[2], and (iii) spatial operation. This section is a reflection of our experience, describing how these operations can be used, as available in existing frameworks, to implement a typical analysis task. In each of the example tasks, we detail why a straight-forward implementation may not be sufficient, alluding to the need for domain specific optimizations.

3.2.1 Sliding Window Operations

Persistent hotspot tracking per sliding window We define a traffic hotspot to be a group of close by base stations, each of whose traffic volume is above a threshold for the time window (referred to as snapshot). We can construct a graph with nodes representing base stations. Two nodes are connected by an edge if and only if the traffic volumes of both nodes exceed the threshold. Detecting hotspots is then equivalent to computing connected components on this graph. Although the computation can be expressed in a distributed dataflow framework [30, 36] using `join` and `group-by` operators, this can be very inefficient as shown in prior work in graph parallel systems [18, 25, 26]. The reason is that join operators are not optimized for graph processing and can be very expensive. Network operators need to compute traffic hotspots per time window. In addition, they also need to detect persistent hotspots (a hotspot is persistent for a sliding window if it is a hotspot in all the component intervals). This task thus requires computation across many windows.

As a concrete example, Figure 2a shows the hotspot graph for three time windows. Suppose we want to compute the persistent hotspots for a sliding window of 3. A straightforward approach is to merge the graphs and perform connected component computation on the resulting graph using a graph parallel processing engine such as GraphLab, GraphX, or GraphLINQ [18, 25, 28]. However, we found that this strategy to be very inefficient when the sliding window is large.

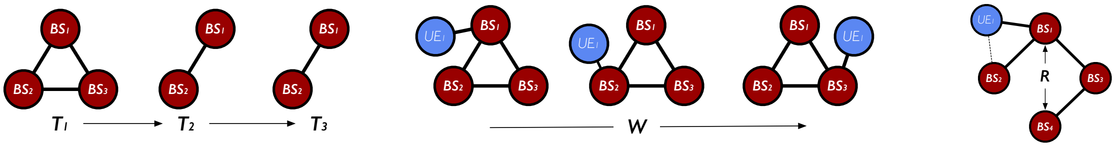
A better approach is to maintain a cumulative graph which counts the number of edges. We then subtract the edge counts from the time window that needs to be forgotten, thus applying *differential* updates to the underlying graph. For the example presented earlier, the edge count for BS1—BS2 is 3, while that for BS2—BS3 and BS1—BS3 is 1. Suppose the graph at time window 0 is empty. We can perform the computation on this cumulative graph. Two nodes are in the same connected component iff their edge count is 3. Hence, the persistent hotspot is BS1—BS2. We demonstrate that this technique speeds up sliding window operations by up to 3× (§ 6).

3.2.2 Time Window Operations

Popular handoff sequence tracking per time window Handoffs can cause connection failures or performance degradation. Operators are interested in monitoring popular handoff sequences in time windows and across sliding windows. A handoff sequence is a valid base station traversal sequence by a set of UEs. If we keep track of both the sequence and the set of associated UEs, then handoff sequence tracking can be implemented as an iterative graph algorithm. Consider the example in Figure 2b, where UE1 is handed off from base station BS1 to BS2 and then from BS2 to BS3 over a time window W. If we are interested in computing the popular handoff sequences in this window W, then BS1 sends the ID of UE1 and the sequence it observes, BS1→BS2, to BS2. In the next iteration, BS2 appends the sequence with its observation, and forwards the new sequence, BS1→BS2→BS3, to BS3. A shortcoming of this approach is that the state management and iterations required to converge becomes a bottleneck for large windows. Thus, the analysis slows down significantly.

A simple optimization to this strategy is to divide W into smaller windows w. However, we cannot compute sequences in each of these windows independently and

[2] It may appear that (i) and (ii) are similar, but it is important to note the difference. Sliding window operations requires more state management, since records in one window may be reused in the next. In other words, the *expiry* of records in a sliding window are variable, while that in a time window are same.



(a) Monitoring persistent hotspots.

(b) Popular handoffs in a window.

(c) Traffic gradients.

Figure 2: Representative analysis tasks of interest to cellular network operators.

then combine them[3]. Instead, we bootstrap every window w with the previous window's handoff sequence and *incrementally* update the graph. Applying this technique results in speeding up analysis tasks that depend on time window operations by 2× to 5× (§ 6).

3.2.3 Spatial Operations

Top traffic gradients tracking Users may converge to a particular location. Operators need to predict these movements and re-optimize their network (e.g., self-optimization techniques such as antenna tilt adjustment and interference coordination) to handle such situations. A traffic gradient of a base station is defined as the weighted average of traffic moving towards it. A coarse grained approximation is to consider all handoffs around a distance of a certain radius R. For each handoff, we project the speed towards the base station and weigh by the product of the bearer throughput divided by the distance between the source base station and the base station under consideration. For example, in Figure 2c, suppose there is a handoff of UE1 from BS1 to BS2. The handoff information will propagate to all base stations in a radius R. $BS4$ (not a direct neighbor of either BS1 or BS2) will add the traffic gradient of this handoff to its current gradient. Currently most graph processing systems only propagate message on a hop-by-hop basis, thus this analysis would be inefficient if implemented directly. A better approach is to *broadcast* the message to a multi-hop neighborhood in one iteration. We found this optimization to speed up this analysis by up to 4× (§ 6).

3.3 Solution Overview

The examples we discussed previously show that cellular network analytics systems require a computation model that can process property graph streams efficiently. To achieve this goal, we presented a case for leveraging cellular specific optimizations exploiting spatial and temporal locality. Ideally we would like a single processing engine that can support a combination of incremental data-parallel processing, stream processing and graph-parallel processing. Since the Berkeley Data Analytics Stack (BDAS) [34] supports all of these computation models, we chose to build CellIQ on BDAS. However, we note that the techniques we present are not restricted to a particular framework; for instance, CellIQ's spatial optimization techniques can be incorporated into a different framework such as Naiad [30].

The key abstraction of the BDAS stack is called Resilient Distributed Datasets (RDDs) [36] which can recover data without replication by tracking the lineage graph of operations that were used to build it. GraphX [18], BDAS's graph-parallel engine, builds on top of the RDD abstraction. It represents graph structured data (called property graph) as a pair of vertex and edge property collections (implemented as RDDs). GraphX embeds graph computation within the Spark distributed dataflow frameworks and distill graph computation to a specific join-map-group-by dataflow pattern. It introduces a range of optimizations both in how graphs are encoded as collections and as well as the execution of the common dataflow operators.

CellIQ is implemented as a layer on top of GraphX and incorporates several domain specific optimizations:

- **Data placement** We implement geo-partitioning of the input data. Vertex properties, edge properties and graphs from different snapshots are co-partitioned. This minimizes data movement.
- **Radius based message broadcast** For messages that need to reach a radius of nodes, we enable the exchanges to complete in one iteration.
- **Spatial aggregation** We implement spatial aggregation for tasks that depend on aggregate statistics such as intra-tracking and inter-tracking area handoff monitoring.
- **Differential graph updates** Tasks that require sliding window operations are implemented using differential updates to the underlying graph over the time windows under consideration.
- **Incremental graph updates** Time window operations are optimized using incremental updates.

We wrap these optimizations with a cellular specific programming abstraction, *G-Stream*. The G-Stream API exposes a domain-specific combination of streaming and graph processing. In the rest of the paper, we describe the CellIQ system (§ 4) and the optimizations (§ 5) in detail.

[3] Doing so without extensive state management would entail incorrect results, because computing sequences independently would miss some subsequences that happen across windows.

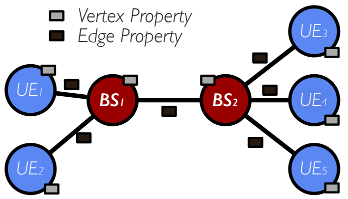


Figure 3: LTE network monitoring data as property graphs

4 CellIQ System

In this section, we describe how CellIQ represents cellular network data, and optimizes the placement for efficient analysis. We then discuss the computational model.

4.1 Graph Representation

Cellular monitoring data as property graphs Our data model for a window of cellular monitoring data is a graph $G(V,E)$, where the vertex is either a user equipment or a base station. For the purpose of the analytics we are interested in, we discard other entities, although it is easy to incorporate them if required. An edge is formed between a user and the base station to which she is connected, Thus, each base station vertex consists of many edges. Similarly, an edge is formed between two base stations when a user traverses between them (i.e., she is handed-off from the first base station to the second). To access the cellular network, a UE performs various procedures during which it exchanges control messages with the network. These procedures are carried out using complex protocols modeled as state machines. Any action the user wishes to take in the network, such as browsing the web, watching a video or making a voice call, triggers a myriad of control plane messages. We incorporate these control plane monitoring records as edge properties. Handoffs records are edge properties of the previous base stations. We do not replicate the record for the new base stations. Essentially, all control plane interaction between a user and a base station is stored on the edge between them. Similarly, the edge between two base stations may store records relevant to user traversals between them. Figure 3 depicts such a simple graph.

This representation enables us to do computations on the control plane data efficiently. For instance, the path of a user can be found using a simple graph traversal. Similarly, aggregate base station information can be obtained without any costly map and shuffle operations.

4.2 Graph Partitioning

A straight-forward approach to distributing the graph in a cluster is to partition the vertices and edges among the machines using a hash-partitioner. Although such a scheme ensures uniform distribution of vertices and edges, it also places neighboring vertices in different machines, thus resulting in poor performance. PowerGraph [19] introduces the vertex-cut technique based on the observation that natural graphs exhibit power law degree distribution, and proposes a greedy edge placement algorithm that minimizes vertex replication. Cellular network graphs do not typically exhibit power law degree distribution and hence the algorithm is not directly applicable. An alternative approach is to use balanced edge-cuts (e.g., as proposed by METIS [23]) for partitioning. However, this results in two disadvantages. First, they result in edge replication which are costly in our graph representation with many thousands of records stored in edges per time window. Second, since edge properties change over time, an edge-cut that is optimal in one snapshot may not remain so in the next, requiring expensive data movements. To strike a balance between these advantages and shortcomings, CellIQ uses the vertex-cut strategy to avoid replicating edges, and a geo-partitioning technique to place edges to preserve spatial locality.

Geo-partitioning of data For efficient analysis of cellular network data, CellIQ requires nodes that are physically close by to be present in the same partition. To achieve this, CellIQ uses a geo-partitioner to distribute the graph across the cluster. The partitioner first maps the vertices to real world geo-coordinates. Each user inherits the location of the base station they are connected to. A standard way to organize multidimensional co-ordinates is to use a tree based datastructure (e.g., Quad-trees [17] or R-trees [20]), but they require complex look ups in a distributed setting. In order to leverage the key based look up schemes typical in cluster computing frameworks, CellIQ's geo-partitioner uses a space-filling curve based approach. The key idea behind space filling curves is to map 2-dimensional locations to 1-dimensional keys that preserve spatial proximity [33]. Thus, keys that are contiguous represent contiguous locations in space. We convert the geo-location of each of the nodes in the graph to its corresponding 1 dimensional space-filling curve key. The key space is then range-partitioned to the machines in the cluster. Edges are co-partitioned with vertices by assigning them the key associated with the source vertex.

Edge indexing Many base station properties such as traffic volume, aggregate frame losses are computed from edge properties between base stations and UEs. To enable fast computation, we index edge properties by base station ID. This ensures the edge properties of a given base station will most likely end up in one partition.

4.3 Computation Model: Discretized Graph Streams (G-Streams)

Because cellular network data arrives continuously, we need to perform the analysis tasks in a streaming fashion. We treat a streaming computation as a series of deterministic batch graph computations on small time intervals (snapshots). The data received in each interval is stored in the cluster to form an input dataset for that interval. Once the time interval completes, this dataset is processed via deterministic graph parallel operations, such as subgraph, connected component, etc to produce new datasets representing either program outputs or intermediate state.

We define *discretized graph streams (G-Streams)* as a sequence of immutable, partitioned datasets (property graphs as a pair of vertex and edge property collections) that can be acted on by deterministic *transformations*. User defined cellular network analytics programs manipulate G-Stream objects. In contrast, D-Streams are defined on a sequence of RDDs instead of property graphs. We will show that our computations cannot be easily expressed using the D-Stream API.

5 CellIQ API and Optimizations

In this section, we present the APIs and various optimization techniques in CellIQ.

5.1 GeoGraph API

The `GeoGraph` represents the domain specific property graph presented in the previous section, and incorporates the spatial optimizations in CellIQ. The methods exposed by the API is shown in Listing 1.

```
class GeoGraph[V, E] extends Graph[V, E]{
    ...
  //For efficient message exchanges
  def sendMsg (radius: Double, V, V) : M

  //For spatial aggregation tasks
  def spatialAG(reduceV: (V, V) => V,
        reduceE: (E, E) => E) : GeoGraph[V, E]) = {
    val superV: Collection[(ccId, V)] =
        this.vertices.groupBy(ccId, reduceV)
    val superE: Collection[(ccId, ccId, E)] =
        this.triplets.map {
             e => (e.src.cc, e.dst.cc, e.attr)}
           .groupBy((e.src.cc, e.dst.cc), reduceE)

    //Return the final graph
    Graph(superV, superE)
  }
}
```

Listing 1: Spatial graph API for cellular monitoring data.

5.1.1 Message Broadcast Within a Radius

Similar to the traffic gradient tracking example presented earlier, many analysis tasks may require messages from a node in the graph to be propagated to every other node within a geographic distance that far exceeds a single hop[4]. GraphX implements this operation using triplets (a triplet contains an edge and its property, and the two component vertex properties), which requires join operations. Since the operation has to be repeated for every iteration (hop), it becomes expensive. The `sendMsg` API is designed to enable efficient message broadcast to multiple nodes rather than just the immediate hop neighbor. It uses a routing table similar to the one maintained by GraphX. These routing tables are maintained in the vertex partitions and identifies the edge partitions that have edges associated with each vertex in the vertex partition.

In CellIQ, the edges are defined by a distance threshold. We decompose the entire space of interest into subspaces using the threshold by overlaying a grid. For each node, we can compute the subset of subspaces that may contain nodes within a radius R. We maintain a subspace to edge partition mapping that enables easy lookup. This approach is much more efficient than the hop-by-hop propagation, as it minimizes the overheads of joins to a constant instead of being proportional to the hop count.

5.1.2 Spatial Aggregation

Many classes of analysis require operations on spatially aggregated graphs. For instance, operators are interested in tracking intra-tracking area and inter-tracking area handoffs. A tracking area consists of a set of base stations. Inter-tracking area handoffs are more involved which consume high signaling resources and are thus more prone to failures. To assist this kind of tasks, CellIQ exposes the `spatialAG` function. The function assumes that each graph vertex contains a field `cc` that can be used for vertex and edge grouping. The function takes two reduce functions: one for aggregating vertex properties and the other for aggregating edge properties.

As an example, to compute inter-tracking area and intra-tracking area handoffs, we could use the tracking area ID (TAI) field as the `cc` field. Our vertex reduce function would sum up each component vertex's property fields such as traffic volume. If we are only interested in handoffs, then this function may return null. For the edge reduce function, we return the total handoffs. Note that we allow self-edges. A self-edge property is the sum of intra-tracking area handoffs.

5.2 GStream API

The `GStream` API in CellIQ is as described in Listing 2. The input to CellIQ is a stream of `GeoGraphs`. Similar to DStreams [37], we implement operations on this streaming domain specific graph by batching their execution in small time steps. In our system, input graph streams are read from the network. Two types of operations apply to these graph streams: (1) *Transformations* create a new G-Stream from one or more parent streams. These can either be *stateless*, applying separately on the property graph in each time interval or stateful, producing states across time intervals. (2) *Output operations*,

[4] In metropolitans, base stations may be placed as close as a few hundred meters from each other, while the analysis may look at areas spanning several miles.

```
class GStream[V, E] extends Serializable {
     ...
def vertexStream(): DStream[(Id, V)] =
    this.map(g => g.vertices)
def edgeStream(): DStream[(Id, Id, E)] =
    this.map(g => g.edges)

def graphReduce(reduceFunc(Graph[V, E],
    Graph[V, E], fv: (V, V) => V), fe: (E, E) => E)
    ): Graph[V, E] =
    this.reduce((a, b) => reduceFunc(a, b, fv, fe))

// Return a new Gstream by reducing the input graph
// over a sliding window
def graphReduceByWindow(
    reduceFunc(Graph[V, E], Graph[V, E],
               fv: (V, V) => V),
               fe: (E, E) => E): Graph[V, E],
    windowDuration: Duration,
    slideDuration: Duration
    ): GStream[V, E] = {
    this.window(windowDuration,
    slideDuration).map(x => x.graphReduce(reduceFunc))
  }
}
```

Listing 2: GStream API

similar to Spark, write data to external systems.

Even though we can not directly extend D-Stream API, we provide two functions that maximally reuse D-Stream functions. The two functions convert a G-Stream into an independent vertex property D-Stream and edge property D-Stream. These can use all the original D-Stream functions, specifically the functions on collections of key value pairs. The key for the vertex D-Stream is the vertex ID and value is the vertex property. Similarly, the key for the edge D-Stream is the edge ID and value is the edge property. Since the individual component RDDs (vertex or edge RDDs) of the D-Stream are geo-partitioned, they automatically take advantage of our spatial optimizations.

G-Streams support the same stateless transformations available in GraphX including subgraph, connected components and join of vertex and edge RDDs. In addition, G-Streams also provide several *stateful* transformations for computations across multiple time intervals.

Windowing: Similar to D-Stream windowing operator, the *window* operation groups all the graphs from a sliding window of past time intervals into one. For example, calling gs.window("5s") yields a G-Stream containing graphs in intervals [0,5), [1,6), [2,7), etc.

graphReduce: Reduces a G-Stream into a GeoGraph.

Sliding window: The *graphReduceByWindow* operation computes one graph per sliding window.

5.2.1 Extending GraphX Operators to Support `graphReduce`

We represent each time window of data as a property graph. To perform window computations, we need to reduce a sequence of graphs into one graph. GraphX does not support certain graph transformations such as intersection and union. We extend the GraphX API to support these transformations. Both `intersection` and `union` operators take two graphs, a vertex function and an edge function. The vertex function decides what to do with the vertex properties of each common vertex. Similarly, the edge function decides how to combine the edge properties of each common edge. An `intersection` operator performs a GraphX `innerJoin` operation on either two vertex or two edge RDDs, and keeps only common vertices and edges in both graphs. A `union` operator performs a GraphX `outerJoin` operation and keeps all vertices and edges from both graphs.

```
def persistConnectedComponents(gs: GStream) = {
    val gs1 = gs.graphReduceByWindow(
          (a, b) => a.intersection(b,
                    (id, V1, V2) => id,
                    (id1, id2, E1, E2) => (id1, id2),
         "1s", "5s")
    val hotspots = gs1.map(_.connectedComponents())
}
```

Listing 3: Connected components in sliding windows.

Listing 3 illustrates the use of the `graphReduce` operator by computing the connected components in each sliding window, where we reduce each sliding window into a graph using the `intersection` operator as the reduce function. We then output the connected component in each sliding window of 5s using the `connectedComponents` operator of GraphX. Similarly, to compute popular handoff sequences for each sliding window, we collect all handoff sequences for each sliding window using the `reduceByWindow` operator. We then sort the sequences by the number of UEs traversing them.

5.2.2 Differential Updates for Sliding Window Operations

For sliding window computation, if we have to perform pair-wise graph reduce operation, it can be very expensive. To enable differential computation, we provide differential aggregation of property graphs. The differential version of `graphReduceByWindow` takes an graph aggregation function and a function for "subtracting" a graph. The incremental computation can be implemented in this framework by using a null subtraction function and then resetting the graph at every window.

In the example shown in listing 4, the graph aggregation function just sums up the vertex count and edge count of two graphs. The subtraction function just subtracts vertex count and edge count of one graph from the other. For each sliding window of K snapshots, instead of computing $K-1$ graph intersections, we only perform one graph union and one graph subtraction. We union the cumulative graph with the graph of the current snapshot, and subtract the graph at $t-K$ time interval where t is the current interval number. To compute the persistent hotspots

```
def persistConnectedComponents(gs: GStream) = {
    val gs1 = gs.graphReduceByWindow(
               (a, b) => a.union(b,
                  (id, V1, V2) => (id, V1.cnt+V2.cnt),
                  (id1, id2, E1, E2)
                         => (id1, id2, E1.cnt+E2.cnt),
               (a, b) => a.intersection(b,
                  (id, V1, V2) => (id, V1.cnt-V2.cnt),
                  (id1, id2, E1, E2)
                         => (id1, id2, E1.cnt-E2.cnt),
               "1s", "5s")

    val hotspots = gs1.map(x =>
               x.subgraph(vPred = (id, c) => c>=K,
                          ePred = (id1, id2, cV1,
                                cV2, cE) => cE>=K)
                .connectedComponents())
}
```

Listing 4: Incrementally computing connected components in each sliding window

for a sliding window, we filter vertices and edges whose count are smaller than K using the subgraph operator of GraphX, and then run connectedComponents.

Similarly, for handoff sequence, we accumulate the list of UEs traversed a handoff sequence (list combine). For subtraction, we just remove the tail elements of the sequence from $t-K$ time interval.

5.3 Co-partitioning Component Graphs

As shown in Chronos [21], in general, it is very hard to accommodate graph structure locality (neighborhood) per snapshot and temporal locality (co-locate vertices or edges in different time windows) across snapshots. Applications or systems have to make a tradeoff between retaining structure locality and temporal locality for evolving graphs. In cellular network data, edges in one snapshot have spatial locality and edges across snapshot retain most of the spatial locality as users do not move long distance over short time windows. As a result, we co-partition all graph snapshots in the active set (old snapshots are cleaned up). This co-partition retains both structural and temporal locality, and significantly reduces data movement for computations on G-Streams.

5.4 Indices and Routing Tables

GraphX maintains indices on the partitions that vertices or edges reside. It also keeps a routing table so that a vertex can find out which edge partitions contain its neighbors. We share the same index and routing data structures for all component graphs in a G-Stream since we co-partition the component graphs.

6 Evaluation

We evaluated CellIQ's performance using the three representative analysis tasks we presented in § 3. Our results are summarized below:

- Geo-partitioning has a significant impact in CellIQ's performance. The improvement due to this partitioning strategy ranges from 2× in small analysis windows to several orders of magnitude in larger windows. In addition, geo-partitioning enables analysis to complete when other partitioning strategies fail due to the data movement overhead.
- CellIQ's incremental graph update strategy results in the reduction of analysis time by 2× to 5×.
- The differential graph update technique significantly benefits sliding window computations, by improving performance by up to 4×. Moreover, the technique enables CellIQ to perform well for various window sizes, when strawman techniques incur increasing performance penalty when the analysis window becomes larger.
- Radius based broadcast improves the analysis time by up to 4× compared to the standard hop-by-hop propagation approach.

We discuss these results in detail in the rest of this section after describing our evaluation set up and the datasets used in our experiments.

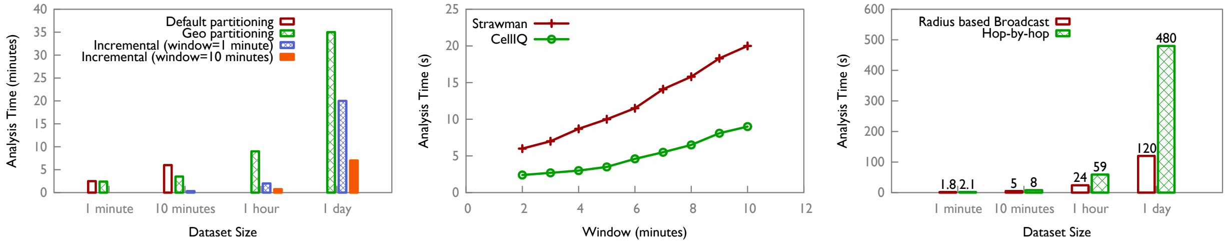
Evaluation Setup: Our evaluation environment consists of 10 machines forming a cluster. Each machine consists of 4 CPUs, 32GB of memory and a 200GB magnetic hard disk. In addition to HDFS, a network storage of 1TB is accessible from all the machines. CellIQ system was built on GraphX version 1.0.

Dataset: We obtained LTE control plane data from a major cellular network operator. The data is from a live network which serves around 1 million subscribers in a large metropolitan area. A single file is generated every minute, and contains around 750,000 records. We receive 10 such files every minute from 10 collection points, bringing the total number of records per minute to approximately 7.5 million. Thus, in the following experiments, we process 450 million records for window sizes of 1 hour and 4.5 billion records[5] for a day window. We store a week worth of data in HDFS, which accounts to approximately 2 terabytes of compressed data.

6.1 Tracking Popular Handoff Sequences

With the increase in base station deployment in an effort to combat the increasing demands in data traffic, handoffs become inevitable when users are mobile even to a small extent. While most handoffs are benign, analyzing handoff patterns often helps operators uncover end-user performance issues. For instance, ping-pong handoffs may indicate an incorrect base station configuration, and unexpected handoff sequences seen by many users may indicate interference issues. The results from this application can be combined with other metrics, such as downlink throughput, to uncover problematic sequences.

[5] The operator collects data only during the 10 most active hours.



(a) Partitioning and incremental updates. **(b)** Differential updates. **(c)** Radius based message broadcast.

Figure 4: Partitioning and incremental update has a significant impact on the analysis time (a missing value in 4a indicates either an invalid analysis such as 10 minute incremental window on 1 minute analysis, or a timeout due to memory issues). Sliding window computations benefit from differential updates. Radius-based broadcast can further improve the performance on large datasets.

We implemented this in CellIQ using a program that closely matches Pregel. The program takes in a window W, and outputs the top N sequences in the window. The program bootstraps by assigning each edge information on the users that traversed them along with their count. The vertices (base stations) book-keep the handoff sequences, initially an empty set. At every iteration, the vertices send messages to their neighbors. The message consists of the users who traversed from the source vertex to the destination vertex. Clearly, the bootstrap message sends all users who were present at the source vertex at window start. Subsequent messages consist of users who reached the source vertex from other vertices. Thus, after k^{th} iteration, each vertex learns about a handoff sequence of length $k+1$. The algorithm converges when there are no more messages to send.

Benefits of geo-partitioning: To understand the benefits of data placement, we ran this program on datasets of varying sizes, namely 1 minute, 10 minutes, 1 hour and 1 day, with two partitioning schemes. The default data placement distributes the edges across machines so as to balance the load[6]. In contrast, CellIQ's geo-partitioner uses the location of the source vertex as the key. The results of the comparison of performance of the two schemes are depicted in figure 4a. The gains of data placement are clear from the results, which indicate improvements ranging from 2× (smaller datasets) to several orders of magnitude (larger sets) for the geo-partitioned case[7]. As expected, we see the benefits increase with the size of the dataset. The primary reason for this behavior is the locality achieved by the partitioner. When nodes that are geographically close by are placed in the same partition, the number of messages that a node needs to send to other partitions are reduce drastically. The reduction closely matches the performance difference.

Benefits of incremental graph updates: Next, to evaluate the performance of the incremental graph update technique, we reran the analysis program with a few small changes. Instead of running the program on the entire window W, we break up W into smaller windows w. Rather than naively running the program every w and then combining the results, CellIQ uses the `graphReduce` operations (§ 5) that maintain the result from every window w. The next window is bootstrapped from this result. The analysis time for running this program on the same dataset is shown in figure 4a.

We were surprised to see the benefits of this strategy, the reduction in analysis time showed a factor of 2× to 5×. Upon closer evaluation, these benefits come from two sources. First, the incremental update limits the amount of graph unions performed to one. Second, when the analysis graph is kept smaller, the number of messages to be sent in each iteration is reduced. An interesting observation is that the performance of the incremental strategy is better with 10 minute window compared to 1 minute window. Increasing the window size to 15 results in a lower performance compared to 10 minutes. We tried experimenting with different window sizes, and found that very small windows tend to have poor performance. Due to the lack of space, we do not present the results. Finding the optimal window size that minimizes the analysis time is beyond the scope of this work, and may be obtained using techniques similar to those detailed in [16].

Benefits of differential graph updates: Finally, to evaluate the efficacy of differential updates on sliding windows, we conducted the following experiment. We streamed one day's data to CellIQ. The handoff analysis is done on this data in batches of 1 minute and slide durations of 1 minute. We varied the window of analysis from 2 minutes to 10 minutes. Thus, a window of 2 minutes indicate that every minute (slide duration), the system computes handoff sequences for the last 2 minute data. The strawman approach keeps a ring buffer where it saves the graph every batch. Then, at the slide window, it combines all the graphs and computes handoff sequences. In

[6] We also used the 2D partitioner in GraphX, but results were similar.

[7] In large datasets, the default partitioner failed to run due to the number of messages generated.

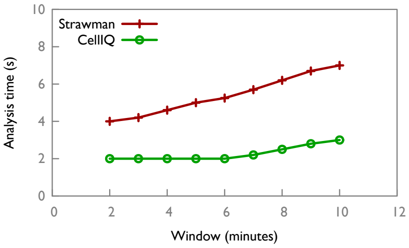


Figure 5: CellIQ's differential update strategy is able to scale simple and complex graph algorithms well.

contrast, CellIQ uses `graphReduceByWindow` that maintains a cumulative graph of the number of users traversing edges. Thus, at every slide window, it only needs to subtract the first graph in the window to obtain the graph on which the analysis needs to be performed. Both approaches use geo-partitioning. The results from this experiment are shown in figure 4b. The strawman approach is able to compute handoffs relatively easily when the window sizes are small. However, when the analysis window is increased, it needs to join many graphs to obtain the result. In comparison, CellIQ is able to scale well due to the fixed number of operations it performs to compute the results. The performance improvement of CellIQ ranged from 2× to 3× in this experiment.

6.2 Monitoring Persistent Hotspots

Arguably, the popular handoff tracking task uses a reasonably complex algorithm that depends on iterative messaging. How does CellIQ's differential update strategy perform on standard graph algorithms that are not message heavy? To answer this question, we implemented an analysis task that continuously monitors hotspots in a given region. As mentioned earlier, hotspot computation can be represented as finding connected components in a graph. Similar to the task before, we use a strawman that builds a graph for every batch and saves it in buffer. At each slide interval, it analyzes all the saved graphs and runs connected components on the union. In contrast, CellIQ leverages its `graphReduceByWindow` operation and then applies connected components on the result.

We again used one day worth of data for this experiment. The hotspot analysis is done on this streaming data in batches of 1 minute and slide durations of 1 minute. We varied the window from 2 minutes to 10 minutes. The average values are presented in figure 5. We find results similar to the popular handoff monitoring task, except that the analysis runs faster because of the lower messaging overhead. Thus, CellIQ's differential update technique benefits all sliding window operations.

6.3 Computing Traffic Gradients

Finally, we evaluate the radius based message broadcast in CellIQ. To do so, we use a task that computes the gradients of base stations in a given interval. As we discussed in § 3, such analysis can be very useful for network operators in the context of optimizing their network. Consider, for instance, a large crowd moving towards an area (e.g., popular events). In these cases, it is desirable to provision additional capacity in the area of gathering. Today, operators need to pre-provision capacity using advance knowledge of the events.

In our program, vertices (base stations) need to send the gradient of their users to their neighbors. The propagation of a message stops when it reaches a neighbor at radius r. While this looks similar to our earlier example of handoff analysis, there is one key difference: the message sent in every iteration is the same. Hence, CellIQ utilizes radius based message broadcast to avoid the penalty associated with multiple iterations. Comparison of this approach against the standard hop-by-hop iterative approach is depicted in figure 4c. The approach performs very well when the input dataset is large, providing gains of up to 4×. Although the number of messages remain same in both approaches, the need to do hop-by-hop propagation impacts the analysis time. Due to low number of messages, smaller datasets can leverage transport layer optimizations that reduce the relative gain. Such optimizations are limited in larger datasets.

7 Discussion

CellIQ leverages domain knowledge to do efficient analysis. A domain focused approach is likely to raise several concerns. We discuss some concerns about CellIQ, focusing on the versatility and generality of its techniques.

How versatile is CellIQ's API?

The representative analyses we present in this paper are the result of our discussions with cellular network operators. Even though our discussion ended with a long list of analysis requirements, we realized that most of them distilled down to one or a combination of the techniques we propose in this work. Thus, we believe that CellIQ is able to accommodate a large set of analysis requirements, and is not restricted to the examples presented here. It is possible that new requirements would need to be accommodated in the future. Since all of our techniques are built on two fundamental datastructures—`GeoGraph` and `GStream`—operators can develop new analysis tasks using these as the building blocks without significant effort.

How general are CellIQ's techniques?

Though CellIQ's primary motivation is to provide timely and efficient cellular network analytics, we believe that the techniques presented in this paper are widely applicable beyond the cellular networks domain. An area of emerging interest is smart-cities, where transportation system optimization is a key challenge. Our graph partitioning (§ 4) and spatial optimization techniques (§ 5) can easily be extended to do traffic analysis on a large scale. Similarly, another domain that has received significant attention recently is the Internet-of-Things (IoT), which also exhibit spatio-temporal characteristics. While our techniques may not carry over to the IoT domain directly[8], we believe that they could be extended to fit the requirements. We envision generalizing the techniques we presented to arbitrary graphs as part of future work.

Can CellIQ be used for real-time feedbacks?

The analysis we discuss in the paper are focused on providing reports—insights and issues in the network—useful for the network operator. A better scenario is to automatically utilize the insights without human intervention. Can CellIQ support such tasks?

Timely processing of data is of prime importance to providing real-time feedbacks. That is, once the data arrives, it is desirable to process it as fast as possible. We designed CellIQ with fast and efficient analysis as key requirements. Such quick analysis enables CellIQ to be useful for providing real-time feedbacks that can be incorporated into the network. Analyzing the efficacy of feedbacks is not within the scope of our work because of the lack of support for configuration change and/or feedback integration in current generation LTE networks. However, with the increasing interest in Self-Organizing Networks (SON), we see this as a venue for future work.

8 Related Work

Cellular network analytics systems Several deployed cellular network analytics system [3, 15] are based on streaming databases. Like other streaming databases such as Aurora, Borealis, STREAM, and Telegraph [6, 8, 10, 12], it is very hard and inefficient to perform iterative graph parallel computations. CellIQ is designed to support real time domain specific streaming graph computations. In addition, these streaming databases use replication or upstream backup for recovery. These mechanisms require complex protocols. In contrast, CellIQ inherits the efficient parallel recovery mechanism from Spark.

Recently cellular network analytics systems have adopted the Hadoop based framework [5, 35]. However, they do not support efficient streaming graph computations. Since CellIQ's G-Stream abstraction unifies batch processing, graph processing and stream processing, it can support a range of processing models. For example, it can combine batch and stream processing by incorporating historical data in the analysis.

Temporal graph analytics systems Most large-scale graph processing systems have focused on static graphs. Some of these systems [19, 25, 32] can operate on multiple graphs independently. They do not expose an API or optimize for operations spanning multiple graphs. There are a couple of notable exceptions [9, 11, 21]. combBLAS [9] represent graphs and data as matrices and support generalized binary operators. Kineograph constructs incremental snapshots of the graph. Chronos optimizes the in-memory layout of temporal graphs and the scheduling of iterative computation on those graphs. Unlike CellIQ, they do not present a general API that supports incremental sliding window computation and graph reduce operations. More importantly, they are not optimized for cellular network analytics. Cellular network graphs present both spatial (i.e. graph structural) and temporal locality. CellIQ is designed specifically to leverage these characteristics to support efficient analysis.

Large-scale streaming Several recent systems [2, 14, 30, 31] support streaming computation with high-level APIs similar to D-Streams. However, they do not support streaming graph processing. One notable exception is the recent announcement of Naiad [30]'s GraphLINQ [28]. GraphLINQ intends to provide rich graph functionality within a general-purpose dataflow framework. Similar to CellIQ, it can operate on streams of vertices and edges. However, it is not optimized for cellular network analytics. As we have shown, optimizations that leverage the characteristics of cellular network can significantly improve performance. The techniques we presented can be incorporated in other frameworks. For instance, our spatial optimizations can benefit GraphLINQ.

9 Conclusion and Future Work

Current cellular networks lack a flexible analytics engine. Existing cellular data analytic systems are either elementary or lack support for real-time analytics. In this paper, we present CellIQ, an efficient cellular analytic system that can support rich analysis tasks. It represents cellular network data as a stream of property graphs. Leveraging domain specific knowledge, CellIQ incorporates a number of optimizations such as geo-partitioning of input data and co-partitioning of vertices and edges to reduce data movements, radius-based message broadcast for efficient spatial operations, incremental graph updates to avoid the cost of frequent joins in time window operations and differential graph updates for efficient sliding window operations. Our evaluations show that these techniques enable CellIQ to perform 2× to 5× faster compared to implementations that do not consider domain specific optimizations.

We see several arenas for future work. We are working on using CellIQ to perform root cause analysis on operational LTE networks. We would also like to explore the possibility of applying CellIQ's techniques on domains other than cellular networks. In this respect, we are working on streaming graph analysis techniques that are applicable to any arbitrary graphs.

[8] CellIQ assumes that the data is human generated in some of its optimizations which may not be always valid in the IoT domain.

Acknowledgments

Part of this work was performed while Anand was at Bell Labs, supported by NSF grant CNS 1218668. We thank Aurojit Panda, Shivaram Venkataraman and other AMPLab members who provided feedback on earlier drafts of this paper. We also thank our shepherd, Suman Banerjee, and our reviewers for their valuable feedback. This research is supported in part by NSF CISE Expeditions Award CCF-1139158, LBNL Award 7076018, and DARPA XData Award FA8750-12-2-0331, and gifts from Amazon Web Services, Google, SAP, The Thomas and Stacey Siebel Foundation, Adatao, Adobe, Apple, Inc., Blue Goji, Bosch, C3Energy, Cisco, Cray, Cloudera, EMC, Ericsson, Facebook, Guavus, Huawei, Informatica, Intel, Microsoft, NetApp, Pivotal, Samsung, Splunk, Virdata and VMware.

References

[1] Aggarwal, B., Bhagwan, R., Das, T., Eswaran, S., Padmanabhan, V. N., and Voelker, G. M. Netprints: diagnosing home network misconfigurations using shared knowledge. In *Proceedings of the 6th USENIX symposium on Networked systems design and implementation* (Berkeley, CA, USA, 2009), NSDI'09, USENIX Association, pp. 349–364.

[2] Akidau, T., Balikov, A., Bekiroğlu, K., Chernyak, S., Haberman, J., Lax, R., McVeety, S., Mills, D., Nordstrom, P., and Whittle, S. Millwheel: Fault-tolerant stream processing at internet scale. *Proc. VLDB Endow. 6*, 11 (Aug. 2013), 1033–1044.

[3] Alcatel Lucent. 9900 wireless network guardian. http://www.alcatel-lucent.com/products/9900-wireless-network-guardian, 2013.

[4] Alcatel Lucent. Alcatel-Lucent motive big network analytics for service creation. http://resources.alcatel-lucent.com/?cid=170795, 2014.

[5] Alcatel Lucent. Motive big network analytics. http://www.alcatel-lucent.com/solutions/motive-big-network-analytics, 2014.

[6] Arasu, A., Babcock, B., Babu, S., Datar, M., Ito, K., Nishizawa, I., Rosenstein, J., and Widom, J. Stream: The stanford stream data manager (demonstration description). In *Proceedings of the 2003 ACM SIGMOD International Conference on Management of Data* (New York, NY, USA, 2003), SIGMOD '03, ACM, pp. 665–665.

[7] Bahl, P., Chandra, R., Greenberg, A., Kandula, S., Maltz, D. A., and Zhang, M. Towards highly reliable enterprise network services via inference of multi-level dependencies. In *Proceedings of the 2007 conference on Applications, technologies, architectures, and protocols for computer communications* (New York, NY, USA, 2007), SIGCOMM '07, ACM, pp. 13–24.

[8] Balazinska, M., Balakrishnan, H., Madden, S. R., and Stonebraker, M. Fault-tolerance in the borealis distributed stream processing system. *ACM Trans. Database Syst. 33*, 1 (Mar. 2008), 3:1–3:44.

[9] Buluç, A., and Gilbert, J. R. The combinatorial BLAS: design, implementation, and applications. *IJHPCA 25*, 4 (2011), 496–509.

[10] Carney, D., Çetintemel, U., Cherniack, M., Convey, C., Lee, S., Seidman, G., Stonebraker, M., Tatbul, N., and Zdonik, S. Monitoring streams: A new class of data management applications. In *Proceedings of the 28th International Conference on Very Large Data Bases* (2002), VLDB '02, VLDB Endowment, pp. 215–226.

[11] Cheng, R., Hong, J., Kyrola, A., Miao, Y., Weng, X., Wu, M., Yang, F., Zhou, L., Zhao, F., and Chen, E. Kineograph: Taking the pulse of a fast-changing and connected world. In *Proceedings of the 7th ACM European Conference on Computer Systems* (New York, NY, USA, 2012), EuroSys '12, ACM, pp. 85–98.

[12] Cherniack, M., Balakrishnan, H., Balazinska, M., Carney, D., Cetintemel, U., Xing, Y., and Zdonik, S. Scalable Distributed Stream Processing. In *CIDR 2003 - First Biennial Conference on Innovative Data Systems Research* (Asilomar, CA, January 2003).

[13] Cohen, I., Goldszmidt, M., Kelly, T., Symons, J., and Chase, J. S. Correlating instrumentation data to system states: a building block for automated diagnosis and control. In *Proceedings of the 6th conference on Symposium on Opearting Systems Design & Implementation - Volume 6* (Berkeley, CA, USA, 2004), OSDI'04, USENIX Association, pp. 16–16.

[14] Condie, T., Conway, N., Alvaro, P., Hellerstein, J. M., Elmeleegy, K., and Sears, R. Mapreduce online. In *Proceedings of the 7th USENIX Conference on Networked Systems Design and Implementation* (Berkeley, CA, USA, 2010), NSDI'10, USENIX Association, pp. 21–21.

[15] Cranor, C., Johnson, T., Spataschek, O., and Shkapenyuk, V. Gigascope: a stream database for network applications. In *Proceedings of the 2003 ACM SIGMOD international conference on Management of data* (New York, NY, USA, 2003), SIGMOD '03, ACM, pp. 647–651.

[16] Das, T., Zhong, Y., Stoica, I., and Shenker, S. Adaptive stream processing using dynamic batch sizing. In *Proceedings of the ACM Symposium on Cloud Computing* (New York, NY, USA, 2014), SOCC '14, ACM, pp. 16:1–16:13.

[17] Finkel, R., and Bentley, J. Quad trees a data structure for retrieval on composite keys. *Acta Informatica 4*, 1 (1974), 1–9.

[18] Gonzalez, J., Xin, R., Dave, A., Crankshaw, D., and Franklin, Stoica, I. Graphx: Graph processing in a distributed dataflow framework. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)* (Broomfield, CO, Oct. 2014), USENIX Association.

[19] Gonzalez, J. E., Low, Y., Gu, H., Bickson, D., and Guestrin, C. Powergraph: Distributed graph-parallel computation on natural graphs. In *Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation* (Berkeley, CA, USA, 2012), OSDI'12, USENIX Association, pp. 17–30.

[20] Guttman, A. R-trees: A dynamic index structure for spatial searching. In *Proceedings of the 1984 ACM SIGMOD International Conference on Management of Data* (New York, NY, USA, 1984), SIGMOD '84, ACM, pp. 47–57.

[21] Han, W., Miao, Y., Li, K., Wu, M., Yang, F., Zhou, L., Prabhakaran, V., Chen, W., and Chen, E. Chronos: A graph engine for temporal graph analysis. In *Proceedings of the Ninth European Conference on Computer Systems* (New York, NY, USA, 2014), EuroSys '14, ACM, pp. 1:1–1:14.

[22] Kandula, S., Mahajan, R., Verkaik, P., Agarwal, S., Padhye, J., and Bahl, P. Detailed diagnosis in enterprise networks. In *Proceedings of the ACM SIGCOMM 2009 conference on Data communication* (New York, NY, USA, 2009), SIGCOMM '09, ACM, pp. 243–254.

[23] Karypis Lab, University of Minesota. Metis - serial graph partitioning and fill-reducing matrix ordering. http://glaros.dtc.umn.edu/gkhome/metis/metis/overview, 2014.

[24] Khanna, G., Yu Cheng, M., Varadharajan, P., Bagchi, S., Correia, M. P., and Verı́ssimo, P. J. Automated rule-based diagnosis through a distributed monitor system. *IEEE Trans. Dependable Secur. Comput. 4*, 4 (Oct. 2007), 266–279.

[25] Low, Y., Gonzalez, J., Kyrola, A., Bickson, D., Guestrin, C., and Hellerstein, J. M. Graphlab: A new framework for parallel machine learning. In *UAI* (2010), P. Gr¨¹nwald and P. Spirtes, Eds., AUAI Press, pp. 340–349.

[26] 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 the 2010 ACM SIGMOD International Conference on Management of Data* (New York, NY, USA, 2010), SIGMOD '10, ACM, pp. 135–146.

[27] McSherry, F., Murray, D. G., Isaacs, R., and Isard, M. Differential dataflow. In *CIDR 2013, Sixth Biennial Conference on Innovative Data Systems Research, Asilomar, CA, USA, January 6-9, 2013, Online Proceedings* (2013).

[28] Microsoft Naiad Team. GraphLINQ: A graph library for naiad. http://bigdataatsvc.wordpress.com/2014/05/08/graphlinq-a-graph-library-for-naiad/, 2014.

[29] MIT Technology Review. How wireless carriers are monetizing your movements. http://www.technologyreview.com/news/513016/how-wireless-carriers-are-monetizing-your-movements/, 2013.

[30] Murray, D. G., McSherry, F., Isaacs, R., Isard, M., Barham, P., and Abadi, M. Naiad: A timely dataflow system. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles* (New York, NY, USA, 2013), SOSP '13, ACM, pp. 439–455.

[31] Qian, Z., He, Y., Su, C., Wu, Z., Zhu, H., Zhang, T., Zhou, L., Yu, Y., and Zhang, Z. Timestream: Reliable stream computation in the cloud. In *Proceedings of the 8th ACM European Conference on Computer Systems* (New York, NY, USA, 2013), EuroSys '13, ACM, pp. 1–14.

[32] Roy, A., Mihailovic, I., and Zwaenepoel, W. X-stream: Edge-centric graph processing using streaming partitions. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles* (New York, NY, USA, 2013), SOSP '13, ACM, pp. 472–488.

[33] Sagan, H. *Space-filling curves*, vol. 18. Springer-Verlag New York, 1994.

[34] UC Berkeley. Berkeley Data Analytics Stack. https://amplab.cs.berkeley.edu/software/, 2014.

[35] Verizon. Verizon adds cloudera's cloud-based big data analytics solution to verizon cloud ecosystem. http://www.verizon.com/about/news/verizon-adds-clouderas-cloudbased-big-data-\analytics-solution-verizon-cloud-ecosystem/, 2013.

[36] Zaharia, M., Chowdhury, M., Das, T., Dave, A., Ma, J., McCauley, M., Franklin, M. J., Shenker, S., and Stoica, I. Resilient distributed datasets: a fault-tolerant abstraction for in-memory cluster computing. In *Proceedings of the 9th USENIX conference on Networked Systems Design and Implementation* (Berkeley, CA, USA, 2012), NSDI'12, USENIX Association, pp. 2–2.

[37] Zaharia, M., Das, T., Li, H., Hunter, T., Shenker, S., and Stoica, I. Discretized streams: Fault-tolerant streaming computation at scale. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles* (New York, NY, USA, 2013), SOSP '13, ACM, pp. 423–438.

Global analytics in the face of bandwidth and regulatory constraints

Ashish Vulimiri[u] *Carlo Curino*[m] *Brighten Godfrey*[u]
Thomas Jungblut[m] *Jitu Padhye*[m] *George Varghese*[m]
[u]*: UIUC <vulimir1,pbg>@illinois.edu* [m]*: Microsoft <ccurino,thjungbl,padhye,varghese>@microsoft.com*

Abstract

Global-scale organizations produce large volumes of data *across* geographically distributed data centers. Querying and analyzing such data as a whole introduces new research issues at the intersection of networks and databases. Today systems that compute SQL analytics over geographically distributed data operate by pulling all data to a central location. This is problematic at large data scales due to expensive transoceanic links, and may be rendered impossible by emerging regulatory constraints. The new problem of Wide-Area Big Data (WABD) consists in orchestrating query execution across data centers to minimize bandwidth while respecting regulatory constaints. WABD combines classical query planning with novel network-centric mechanisms designed for a wide-area setting such as pseudo-distributed execution, joint query optimization, and deltas on cached subquery results. Our prototype, Geode, builds upon Hive and uses 250× less bandwidth than centralized analytics in a Microsoft production workload and up to 360× less on popular analytics benchmarks including TPC-CH and Berkeley Big Data. Geode supports all SQL operators, including Joins, across global data.

1 Introduction

Organizations operating at global scale need to analyze vast amounts of data. The data is stored in multiple data centers around the world because of stringent latency requirements for user-facing applications. The volume of data collected while logging user interactions, monitoring compute infrastructures, and tracking business-critical functions is approaching petabytes a day. These new global databases *across* data centers — as opposed to traditional parallel databases [4] *within* a data center — introduce a new set of research issues at the intersection of databases and networks, combining the traditional problems of databases (e.g., query planning, replication) with the challenges of wide area networks (e.g., bandwidth limits, multiple sovereign domains [17]). Recent work in global databases illustrates this research trend, from Spanner [12] (global consistency) to Mesa [22] (replication for fault tolerance) to JetStream [34] (analytics for data structured as OLAP cubes).

In these applications, besides the many reads and writes generated by user transactions or logging, data is frequently accessed to extract insight, using ad-hoc and recurrent analytical queries. Facebook [40, 44], Twitter [29], Yahoo! [14] and LinkedIn [3] report operating pipelines that process tens or hundreds of TBs of data each day. Microsoft operates several large-scale applications at similar scales, including infrastructures for collecting telemetry information for user-facing applications, and a debugging application that queries error reports from millions of Windows devices [19]. Many of these queries require Joins and cannot be supported using the OLAP cube abstraction of [34].

To the best of our knowledge, companies today perform analytics across data centers by transferring the data to a central data center where it is processed with standard single-cluster technologies such as relational data warehouses or Hadoop-based stacks. However, for large modern applications, the centralized approach transfers significant data volumes. For example, an analytics service backing a well known Microsoft application ingests over 100 TB/day from multiple data centers into a centralized analytics stack. The total Internet bandwidth crossing international borders in 2013 was 100 Tbps (Figure 1). Even if all this capacity were dedicated to analytics applications and utilized with 100% efficiency, it could support only a few thousand such applications.

Moreover, while application demands are growing from 100s of terabytes towards petabytes per day, network capacity growth has been decelerating. The 32% capacity growth rate in 2013-2014 was the lowest in the past decade (Figure 1). A key reason is the expense of adding network capacity: for instance, a new submarine cable connecting South America and Europe is expected

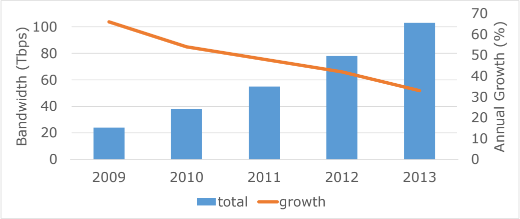


Figure 1: Sum of capacities of Internet links crossing international borders [24]

to cost $185 million. This scarcity of wide-area network bandwidth can drive applications to discard valuable data; the problem will only worsen as applications scale up and out. Our analysis of bandwidth trends is consistent with [34, 28, 21].

A second emerging difficulty is that privacy concerns (for example in the EU [16]) may result in more regulatory constraints on data movement. However, while local governments may start to impose constraints on raw data storage [35], we speculate that *derived* information, such as aggregates, models, and reports (which are critical for business intelligence but have less dramatic privacy implications) may still be allowed to cross geographical boundaries.

Thus our central thesis is: rising global data and scarce trans-oceanic bandwidth, coupled with regulatory concerns, will cause an inflection point in which centralizing analytics (the norm today) will become inefficient and/or infeasible. We consider the problem of providing wide area analytics while minimizing bandwidth over geo-distributed data structured as SQL tables, a dominant paradigm. We support the entire array of SQL operators on global data including Joins, providing exact answers. We refer to such analytics as *Wide-Area Big Data* or WABD.

Our paper proposes a solution to the WABD problem. We support SQL analytics on geo-distributed data, providing automated handling of fault-tolerance requirements and using replicated data to improve performance whenever possible. We assume that resources within a single data center (such as CPU and storage) are relatively cheap compared to cross-data center bandwidth. We target the batch analytics paradigm dominant in large organizations today [44, 29, 3], where the cost of supporting analytics execution is the primary metric of interest. Our optimizations are all targeted at reducing bandwidth cost; we currently make no attempt at minimizing analytics execution latency.

Our techniques revisit the classical database problem of query planning while adding a networking twist. In particular, while classical query planning optimizes query processing ("What's the best join order?") to minimize *computation*, in WABD we optimize the execution strategy to minimize *bandwidth* and respect sovereignity. For example, one of our techniques is based on caching previous answers to subqueries at data centers and only sending the difference, reminiscent of differential file transfer mechanisms [42, 41]. Similarly, our query optimization approach relies on the fact that analytical queries are repeated: thus simple measurement techniques common in networking can be used to measure data transfer costs across data centers. This contrasts with classical database techniques using histograms (designed to handle arbitrary queries) that are well known to be inaccurate in the face of Joins and User Defined Functions [30].

We make four main contributions:

1. *Optimizer:* We jointly optimize query execution plans and data replication to minimize bandwidth cost. Our solution combines a classical centralized SQL query planner (a customized version of Apache Calcite) with an integer program for handling geo-distribution.

2. *Pseudo-distributed measurement:* We develop a technique that *modifies* query execution to collect accurate data transfer measurements, potentially increasing the amount of (cheap) computation within individual data centers, but never worsening (expensive) cross-data center bandwidth.

3. *Subquery Deltas:* We take advantage of the cheap storage and computation within individual data centers to aggressively cache all intermediate results, using them to eliminate data transfer redundancy using deltas.

4. *Demonstrated Gains:* Our prototype, Geode, is built on top of the popular Hive [39] analytics framework. Geode achieves a 250× reduction in data transfer over the centralized approach in a standard Microsoft production workload, and up to a 360× improvement in a range of scenarios across several standard benchmarks, including TPC-CH [9] and Berkeley Big Data [6].

2 Approach Overview

We start by discussing an example inspired by the Berkeley Big-Data Benchmark [6] and use it to motivate our architecture.

Running example

We have a database storing batch-computed page metadata and a log of user visits to web pages, including information about the revenue generated by each visit:

```
ClickLog(sourceIP,destURL,visitDate,adRevenue,...)
PageInfo(pageURL,pageSize,pageRank,...)
```

Pages are replicated at multiple edge data centers, and users are served the closest available copy of a page. Visits are logged to the data center the user is served from, so that the ClickLog table is naturally partitioned across edge data centers. The PageInfo table is stored centrally in a master data center where it is updated periodically by an internal batch job.

Now consider an analytical query reporting statistics for users (identified by their IP address) generating at least $100 in ad revenue.

```
Q: SELECT sourceIP, sum(adRevenue), avg(pageRank)
   FROM ClickLog cl JOIN PageInfo pi
     ON cl.destURL = pi.pageURL
   WHERE pi.pageCategory = 'Entertainment'
   GROUP BY sourceIP
   HAVING sum(adRevenue) >= 100
```

Supporting this query via the centralized approach requires retrieving all updates made to the ClickLog table to a central data center where the analytical query is computed. This means that the daily network bandwidth requirement is proportional to the total size of the updates to the database. Assuming 1B users, 6 pages visited per user, 200 bytes per ClickLog row, this is roughly (1B * 6 * 200) bytes = 1.2 TB per day.

By contrast, Geode provides an equivalent location-independent [33] query interface over distributed data. The analyst submits the query Q unmodified to Geode, which then automatically partitions the query and orchestrates distributed execution. Geode constructs the distributed plan in two stages:

1. Choose join order and strategies. Geode first creates a physical execution plan for the logical query Q, explicitly specifying the order in which tables are joined and the choice of distributed join algorithm for processing each join (broadcast join, semijoin etc. — see §5). In this simple query, there is only one choice: the choice of algorithm for processing the join between the ClickLog and PageInfo tables.

To make these choices we use Calcite++, a customized version we built of the Apache Calcite centralized SQL query planner. Calcite has built-in rules that use simple table statistics to optimize join ordering for a given query; Calcite++ extends Calcite to also make it identify the choice of distributed join algorithm for each join. We describe Calcite++'s design in detail in §4.1.

When Calcite++ is run on `Q`, it outputs an annotation `JOINHINT(strategy = right_broadcast)`, indicating that the join should be executed by broadcasting the (much smaller) PageInfo table to each data center holding a partition of the (larger) ClickLog table, then computing a local join at each of these data centers.

Assuming an organization that operates across three edge data centers, the physical plan Q_{opt} translates directly into the DAG in Figure 2. Each circle is a task: a SQL query operating on some set of inputs. Edges show data dependencies. Tasks can read base data partitions (e.g. q_1) and/or outputs from other tasks (e.g. q_5) as input. All the inputs a task needs must either already be present or be copied over to any data center where it is scheduled. While we do not consider them in this simple example, regulatory restrictions may prohibit some partitions from being copied to certain data centers, thus constraining task scheduling.

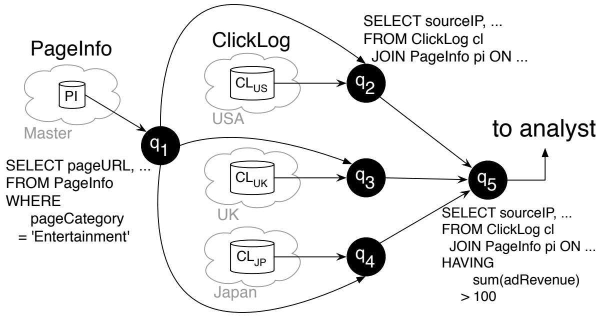


Figure 2: DAG corresponding to Q_{opt}

2. Schedule tasks. Geode now needs to assign tasks to data centers, taking into account task input dependencies and base data regulatory constraints.

Geode can maintain multiple copies of base data partitions, for performance and/or for fault tolerance, and potentially schedule multiple copies of tasks operating on different partition copies. For instance, it could maintain a synchronized copy of the PageInfo table at every data center and create multiple copies of task q_1 at each data center. The choice of replication strategy is controlled by a *workload optimizer*, at a much longer time scale than one individual query's execution (typically replication policy changes occur on a weekly basis or even slower). The optimizer chooses the replication policy taking various factors into account (§4).

At runtime, Geode schedules tasks for individual queries on data centers by solving an integer linear program (ILP) with variables $x_{td} = 1$ iff a copy of task t is scheduled on data center d. The constraints on the ILP specify the input dependencies for each task and the availability and regulatory constraints on copies of partitions at each data center. The ILP tries to minimize the total cost of the data transfers between tasks in the DAG if measurements of inter-task transfer volumes are available (see §4). The ILP described here is a simpler version of the more nuanced multi-query optimizer in §4.3.

Assume an initial setup where data are not replicated. Then the natural strategy is to schedule q_1 on the master data center holding the PageInfo table, push q_2, q_3, q_4 down to the edge data centers holding the ClickLog partition they operate on, and co-locate q_5 with one of q_2, q_3 or q_4. If query `Q` is submitted once a day, 1B users visit 100M distinct pages each day, 100K users have an ad revenue larger than $100, each tuple output by q_1 is 20 bytes long and by q_2, q_3, q_4 is 12 bytes long, distributed execution will transfer $3 * 100M * 20 + (2/3) * 1B * 12 + 100K * 12 = 14$ GB of data each day, compared to 1.2 TB per day for the centralized approach.

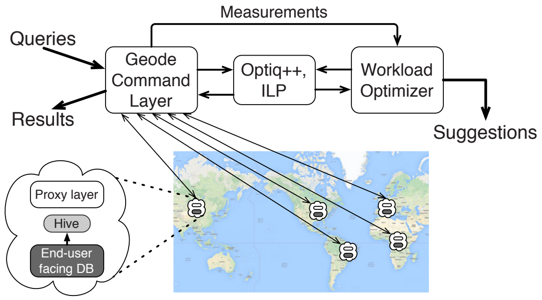


Figure 3: Geode architecture

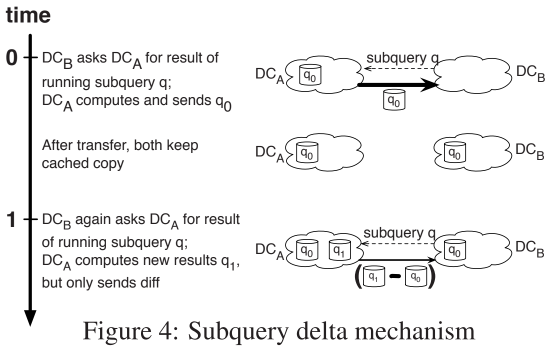

time
0
DC_B asks DC_A for result of running subquery q;
DC_A computes and sends q_0
DC_A
q_0
subquery q
DC_B
q_0
After transfer, both keep cached copy
DC_A
q_0
q_0
DC_B
1
DC_B again asks DC_A for result of running subquery q;
DC_A computes new results q_1, but only sends diff
DC_A
q_0
q_1
subquery q
q_0
DC_B
(q_1 - q_0)

Figure 4: Subquery delta mechanism

While these numbers suggest a clear win for the distributed approach, if Q is submitted once every 10 minutes centralization is more efficient. The workload optimizer evaluates this tradeoff across the entire analytical workload and continuously adapts, reverting to centralized execution if needed. Analytical queries can be much more complex than Q; for example, the CH benchmark (§6) contains a query with 8 joins (involving 9 different tables) for which the degrees of freedom (join order, join strategy, replication) are much higher.

Architecture

Our example motivates the architecture in Figure 3.

Geode processes analytics over data split across multiple data centers, constantly updated by interactions with a set of end-users. End-user interactions are handled externally to our system, and we do not model them explicitly. We assume that at each data center all data has been extracted out into a standard single-data-center analytics stack, such as Hive or a relational database. Our current implementation is Hive-based.

The core of our system is a central command layer. The command layer receives SQL analytical queries, partitions them to create a distributed query execution plan, executes this plan (which involves running queries against individual data centers and coordinating data transfers between them), and collates the final output. At each data center the command layer interacts with a thin proxy deployed over the local analytics stack. The proxy layer facilitates data transfers between data centers and manages a local cache of intermediate query results used for the data transfer optimization in §3.

A workload optimizer periodically obtains measurements from the command layer to estimate if changing the query plan or the data replication strategy would improve overall performance. These measurements are collected using our pseudo-distributed execution technique (§4.2), which may entail rewriting the analytical queries. The optimizer never initiates changes directly, but instead makes suggestions to an administrator.

We next discuss: an optimization we implement to reduce data transfers (§3); the workload optimizer including pseudo-distributed execution (§4); and the design and implementation of our Geode prototype (§5).

3 Subquery deltas: Reducing data transfer

We first turn our attention to optimizing the mechanics of data movement. The unique setting we consider, in which each node is a full data center with virtually limitless CPU and storage, but connectivity among nodes is costly/limited, lends itself to a novel optimization for eliminating redundancy.

Consider a query computing a running average over the revenue produced by the most revenue generating IPs over the past 24 hours. If the query is run once an hour, more than 95% of the data transfer will be wasted because every hour unoptimized Geode would recompute the query from scratch, transferring all the historical data even though only the last hour of data has changed.

We leverage storage and computation in each data center to aggressively cache intermediate results. Figure 4 details the mechanism. After data center DC_B retrieves results for a query from data center DC_A, both the source and the destination store the results in a local cache tagged with the query's signature. The next time DC_B needs to retrieve results for the same query from DC_A, DC_A recomputes the query again, but instead of sending the results afresh it computes a diff (delta) between the new and old results and sends the diff over instead.

Note that DC_A still needs to recompute the results for Q the second time around. Caching does not reduce intra-data-center computation. Its purpose is solely to reduce data transfer between data centers.

We cache results for individual sub-queries run against each data center, not just for the final overall results returned to the analyst. This means that caching helps not only when the analyst submits the same query repeatedly, but also when two different queries use results from the same common sub-query. E.g. in the TPC-CH benchmark that we test in §6, 6 out of the 22 analytical queries that come with the benchmark perform the same join operation, and optimizing this one join alone allows caching to reduce data transfer by about 3.5×.

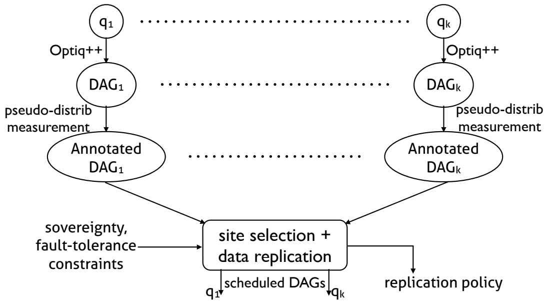

q_1
q_k
Optiq++
Optiq++
DAG_1
DAG_k
pseudo-distrib measurement
pseudo-distrib measurement
Annotated DAG_1
Annotated DAG_k
sovereignty, fault-tolerance constraints
site selection + data replication
replication policy
q_1
scheduled DAGs
q_k

Figure 5: Optimizer architecture

4 Workload Optimizer

Geode targets analytics with a small, slowly evolving core of recurring queries. This matches our experience with production workloads at Microsoft, and is consistent with reports from other organizations [44, 3, 29]. The workload optimizer tailors policy to maximize the performance of this core workload, jointly optimizing:

1. **Query plan**: the execution plan for each query, deciding e.g. join order and the execution mechanism (broadcast join, semijoin etc.).
2. **Site selection**: which data center is used to execute each sub-task for each query.
3. **Data replication**: where each piece of the database is replicated for performance/fault-tolerance.

The problem we face is akin to distributed database query planning. In that context, it is common [27] to employ a two-step solution: (1) find the best centralized plan (using standard database query planning), and (2) decompose the centralized plan into a distributed one, by means of heuristics (often employing dynamic programming). Our approach is similar in spirit, but is faced with substantially different constraints and opportunities arising from the WABD setting:

1. *Data Birth:* We can replicate data partitions to other data centers, but have no control over where data is generated originally – base data are naturally "born" in specific data centers dictated by external considerations, such as the latency observed by end-users.
2. *Sovereignty:* We must deal with the possibility of sovereignty constraints, which can limit where data can be replicated (e.g. German data may not be allowed to leave German data centers).
3. *Fixed Queries:* We can optimize the system for a small, approximately static core workload, which means we do not have to use general-purpose approximate statistics (e.g., histograms) that yield crude execution cost estimates for one-time queries. We can instead collect narrow, precise measures for a fixed core of queries.

These features drive us to the architecture in Figure 5. Briefly, we start by identifying the optimal centralized plan for each query in the core workload using the *Calcite++* query planner (§4.1). We then collect precise measures of the data transfers during each step of distributed execution for these plans using *pseudo-distributed measurement* (§4.2). We finally combine all these measurements with user-specified data sovereignty and fault tolerance requirements to jointly solve the *site selection* and *data replication* problems (§4.3).

4.1 Centralized query planning: Calcite++

Apache Calcite is a centralized SQL query planner currently being used or evaluated by several projects, including Hive [39]. Calcite takes as input a SQL query parse tree along with basic statistics on each table, and produces a modified, optimized parse tree. Calcite++ extends Calcite to add awareness of geo-distributed execution.

Calcite optimizes queries using simple statistics such as the number of rows in each table, the average row size in each table, and an approximate count of the number of distinct values in each column of each table. All these statistics can be computed very efficiently in a distributed manner. Calcite uses these statistics along with some uniformity assumptions to optimize join order. In Calcite++ we leave the join order optimization unchanged but introduce new rules to compare the cost of various (distributed) join algorithms, passing in as additional input the number of partitions of each table. The output of the optimization is an optimized join order annotated with the lowest cost execution strategy for each join — e.g., in our running example (§2) Calcite++ chooses a *broadcast join*, broadcasting PageInfo to all ClickLog locations where local partial joins are then computed.

While both Calcite and (therefore) Calcite++ currently use only simple, rough statistics to generate estimates, in all the queries we tested in our experimental evaluation (§6.1), we found that at large multi-terabyte scales the costs of the distributed join strategies under consideration were orders of magnitude apart, so that imprecision in the generated cost estimates was inconsequential. (The centralized plan generated by Calcite++ always matched the one we arrived at by manual optimization.) Moreover, Calcite is currently under active development — for instance, the next phase of work on Calcite will add histograms on each column.

4.2 Pseudo-distributed execution

The crude table statistics Calcite++ employs suffice to compare high-level implementation choices, but for making site selection and data replication decisions we require much better accuracy in estimating the data transfer cost of each step in the distributed execution plan.

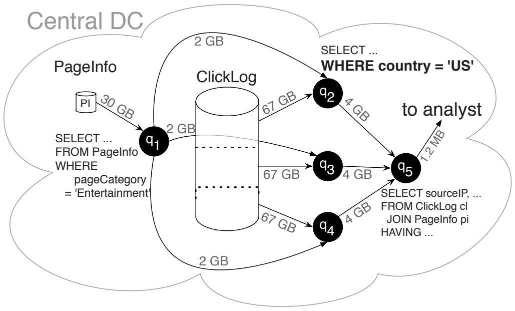


Figure 6: Pseudo-distributed execution of query `Q` (§2) in a centralized deployment. Cf. Figure 2

Traditional database cardinality estimation techniques can be very inaccurate at generating absolute cost estimates, especially in the face of joins and user-defined functions [30]. The sheer volume of data, heterogeneity network topologies and bandwidth costs, and cross-query optimizations such as the sub-query delta mechanism we propose, further complicate statistics estimation.

Instead, we *measure* data transfers when executing the plan in the currently deployed configuration (which could be a centralized deployment or an already running Geode deployment), modifying query execution when necessary to make it possible to collect the estimates we need. As an example consider query Q (Figure 2) from §2, currently running in a centralized configuration (i.e. the entire database is replicated centrally). To estimate the cost of running in a distributed fashion Geode simulates a virtual topology in which each base data partition is in a separate data center. This is accomplished by rewriting queries to push down `WHERE country = X` clauses constraining each of q_2, q_3, q_4 to operate on the right subset of the data[1]. Figure 6 depicts this process. The artificial decomposition allows us to inspect intermediate data sizes and identify the data transfer volume along each edge of the DAG in Figure 2.

This technique, which we call *pseudo-distributed execution*, is both fully general, capable of rewriting arbitrary SQL queries to simulate any given data partitioning configuration, and highly precise, since it directly executes rewritten queries and measures output and input sizes instead of attempting any estimation. We employ the technique whenever we need to evaluate an alternative deployment scenario, such as when considering moving from an initial centralized deployment to a distributed Geode deployment; or when considering adding or decommissioning data centers in a distributed deployment in response to changes in the load pattern.

The latency overhead added by pseudo-distribution is minimal and easily mitigated, as we discuss in §6.2.

Trading precision for overhead: While Geode's implementation of pseudo-distributed execution measures the costs of most SQL queries accurately, including those with joins and nested queries, we deliberately introduce a limited degree of imprecision when evaluating aggregate functions to reduce measurement overhead.

Specifically, we ignore the possibility of partial aggregation within data centers. As an example, suppose 10 data partitions are all replicated to one data center, and consider a `SUM` query operating on all this data. Retrieving one total `SUM` over all 10 partitions is sufficient, but Geode always simulates a fully distributed topology with each partition in a separate data center, thus retrieving separate `SUM`s from each partition and overestimating the data transfer cost. To measure the true cost of function evaluation with partial aggregation we would need an exponential number of pseudo-distributed executions, one for each possible way of assigning or replicating partitions across data centers; one execution suffices for the upper bound we use instead.

We found this was not an issue in any of the workloads (production or benchmark) we tested. The majority of the data transfers during query execution arise when joining tables, and data transfer during the final aggregation phase after the joins have been processed is comparatively much smaller in volume. In all six of our workloads, the data transfer for tasks involved in computing combinable aggregates was $< 4\%$ of the total distributed execution cost.

4.3 Site Selection and Data Replication

After identifying the logical plan (DAG of tasks) for each query (§4.1) and measuring the data transfer along each edge (§4.2), we are left with two sets of decisions to make: *site selection*, specifying which data centers tasks should be run on and which copies of the data they should access; and *data replication*, specifying which data centers each base data partition should be replicated to (for performance and/or fault tolerance). This should be done while respecting disaster recovery requirements and sovereignty constraints.

We formulate an integer linear program (Figure 7a) that jointly solves both problems to minimize total bandwidth cost. The ILP is built from two sets of binary variables, x_{pd} indicating whether partition p is replicated to data center d; and y_{gde} identifying the (source, destination) data center pairs (d,e) to which each edge g in the considered DAGs is assigned[2]. Constraints specify sovereignty and fault-tolerance requirements.

While the ILP provides very high quality solutions, its complexity limits the scale at which it can be applied—as

[1] Every partitioned table in Geode has a user-specified/system-generated field identifying the partition each row belongs to (§5).

[2] We schedule *edges* instead of *nodes* because (1) replication turns out to be easier to handle in an edge-based formulation, and (2) the node-based formulation would have a *quadratic* (not linear) objective.

Inputs:

D = number of data centers

P = number of data partitions

$G = \langle V, E \rangle$ = union of DAGs for all core workload queries

b_g = number of bytes of data transferred along each edge $g \in E$ (from pseudo-distrib. exec.)

update_rate_p = rate at which partition p is updated by OLTP workload (bytes per OLAP run)

link_cost_{de} = cost (\$/byte) of link connecting DCs d and e

f_p = minimum number of copies of partition p that the system *has* to make for fault-tolerance

$R \subseteq P \times D = \{(p,d) \mid$ partition p cannot be copied to data center d due to regulatory constraints$\}$

Variables:

All variables are binary integers (= 0 or 1)

$x_{pd} = 1$ iff partition p is replicated to DC d

$y_{gde} = 1$ iff edge g in the DAGs is assigned source data center d and destination data center e

$z_{td} = 1$ iff a copy of task t in the DAGs is assigned to data center d

Solution:

$$\text{replCost} = \sum_{p=1}^{P} \sum_{d=1}^{D} \text{update\_rate}_p * x_{pd} * \text{link\_cost}_{homeDC(p),d}$$

$$\text{execCost} = \sum_{g \in E} \sum_{d=1}^{D} \sum_{e=1}^{D} y_{gde} * b_g * \text{link\_cost}_{de}$$

$$\underset{X,Y}{\text{minimize}} \qquad \text{replCost} + \text{execCost}$$

subject to

$$\forall (p,d) \in R : x_{pd} = 0$$

$$\forall p : \sum_d x_{pd} \geq f_p$$

$$\forall d \forall e \forall g \mid src(g) \text{ is a partition} : y_{gde} \leq x_{src(g),d}$$

$$\forall d \forall e \forall g \mid src(g) \text{ is a task} : y_{gde} \leq z_{src(g),d}$$

$$\forall n \forall e \forall g \mid dst(g) = n : z_{ne} = \sum_d y_{gde}$$

$$\forall n \forall p \forall d \mid n \text{ reads from partition } p \wedge (p,d) \in R : z_{nd} = 0$$

$$\forall n : \sum_d z_{nd} \geq 1$$

(a) Integer Linear Program jointly solving both problems

```
for all DAG $G \in$ workload do
    for all task $t \in$ toposort($G$) do
        for all data center $d \in$ legal_choices(t) do
            cost(d) = total cost of copying all of $t$'s inputs to $d$
        if lowest cost is zero then
            assign copies of $t$ to every data center with cost = 0
        else
            assign $t$ to one data center with lowest cost
for all $(p,d) \notin R$ do
    check if replicating $p$ to $d$ would further reduce costs
translate decisions so far into values for $x$, $y$, $z$ variables in ILP above
solve simplified ILP with pinned values
```

(b) Greedy heuristic

Figure 7: Site selection + Data replication: Two solvers

we show in §6.3. For example, if we bound our optimization time to 1h (recall that this is an offline, workload-wide process), the ILP can only support up to 10 data centers for workloads of the size we test in our experiments. This is barely sufficient for today's applications. But the rapid growth in infrastructure [21] and application scales that is the norm today may soon outstrip the capabilities of the ILP. As future-proofing, we propose an alternative *greedy heuristic* (Figure 7b) that has much better scalability properties.

The heuristic approach first uses a natural greedy task placement to solve the site selection problem in isolation: identify the set of data centers to which each task can be assigned based on sovereignity constraints over its input data, and greedily pick the data center to which copying all the input data needed by the task would have the lowest cost. We are still left with the NP-hard problem of finding the best replication strategy subject to fault-tolerance and sovereignty requirements. This is tackled by a (much simpler) ILP in isolation from site selection.

The greedy heuristic scales much better than the ILP, identifying solutions in less than a minute even at the 100 data center scale. However, in some cases this can come at the cost of identifying significantly sub-optimal solutions. We evaluate the tradeoff between processing time and solution quality in §6.3.

Limitation: At this point the formulation does not attempt to account for gains due to cross-query caching (the benefit due to the mechanism in §3 when different queries share common sub-operations). The precise effect of cross-query caching is hard to quantify, since it can fluctuate significantly with variations in the order and relative frequency with which analytical queries are run. Similar to the discussion of partially aggregatable functions in the previous subsection, we would need an exponential number of pseudo-distributed measurements to estimate the benefit from caching in every possible combination of execution plans for different queries.

However, we do account for *intra*-query caching — the benefit from caching within individual queries (when the same query is run repeatedly). We always collect pseudo-distributed measurements with a warm cache and report stable long-term measurements. This means all data transfer estimates used by the ILP already account for the long-term effect of intra-query caching.

5 Geode: Command-layer interface

Geode presents a logically centralized view over data partitioned and/or replicated across Hive instances in multiple data centers. Users submit queries in the SQL-like Hive Query Language (HQL) to the command layer, which parses and partitions queries to create a distributed execution plan as in §2. We discuss the basic interface Geode presents to analysts in this section.

Describing schema and placement

Geode manages a database consisting of one or more tables. Each table is either *partitioned* across several data centers, or *replicated* at one or more data centers. Partitioned tables must have a specified *partition column* which identifies which partition any row belongs to. The partition column is used to, among other things, support pseudo-distributed execution and to automatically detect and optimize joins on co-partitioned tables. Partitioned tables can either be *value-partitioned*, meaning each distinct value of the partition column denotes a separate partition, or *range-partitioned* on an integer column, meaning each partition corresponds to a specified range of values of the partition column.

Analysts inform Geode about table schema and placement by submitting `CREATE TABLE` statements annotated with placement type and information — we omit the details of the syntax.

Supported queries

We support most standard analytics features in Hive 0.11 (the latest stable version when we started this project): nested queries, inner-, outer- and semi-joins, and user-defined aggregate functions; although we do not support some of Hive's more unusual feature-set, such as compound data structures and sampling queries [39]. Our architecture is not tied to Hive and can be easily adapted to work with other SQL backends instead.

Joins. By default, Geode passes user-submitted queries through Calcite++ (§4.1) first to optimize join order and execution strategy. However, users can enforce a manual override by explicitly annotating joins with a `JOINHINT(strategy = _)` instruction.

Geode currently supports three classes of distributed join execution strategies: (1) *co-located joins*, which can be computed without any cross-data center data movement either because both tables are co-partitioned or because one table is replicated at all of the other table's data centers; (2) left or right *broadcast joins*, in which one table is broadcast to each of the other table's data centers, where separate local joins are then computed; and (3) left or right *semi-joins*, in which the set of distinct join keys from one table are broadcast and used to identify and retrieve matches from the other table. We are currently exploring adding other strategies, such as hash-joins with a special partitioning-aware hash function [38].

Nested queries. Nested queries are processed recursively[3]. The system pushes down nested queries completely when they can be handled entirely locally, without inter-data-center communication; in this case the results of the nested query are stored partitioned across data centers. For all other queries, the final output is merged and stored locally as a temporary table at the master data center (hosting the Geode command layer). The results of nested queries are transferred lazily to other data centers, as and when needed to execute outer queries.

User-defined functions. We support Hive's pluggable interface for both simple user-defined functions (UDFs), which operate on a single row at a time, and for user-defined aggregate functions (UDAFs). Existing user code can run unmodified.

For UDAFs, note that the need is to allow users to write functions that process data distributed over multiple machines. Hive's solution is to provide a MapReduce-like interface in which users define (1) a *combine* function that locally aggregates all data at each machine, and (2) a *reduce* function that merges all the *combine*d output to compute the final answer. By default we use this interface in an expanded hierarchy to compute UDAFs by applying *combine* a second time in between steps (1) and (2) above, using it on the *combine*d output from each machine to aggregate all data within one data center before passing it on to *reduce*. Users can set a flag to disable this expansion, in which case we fall back to copying all the input to one data center and running the code as a traditional Hive UDAF.

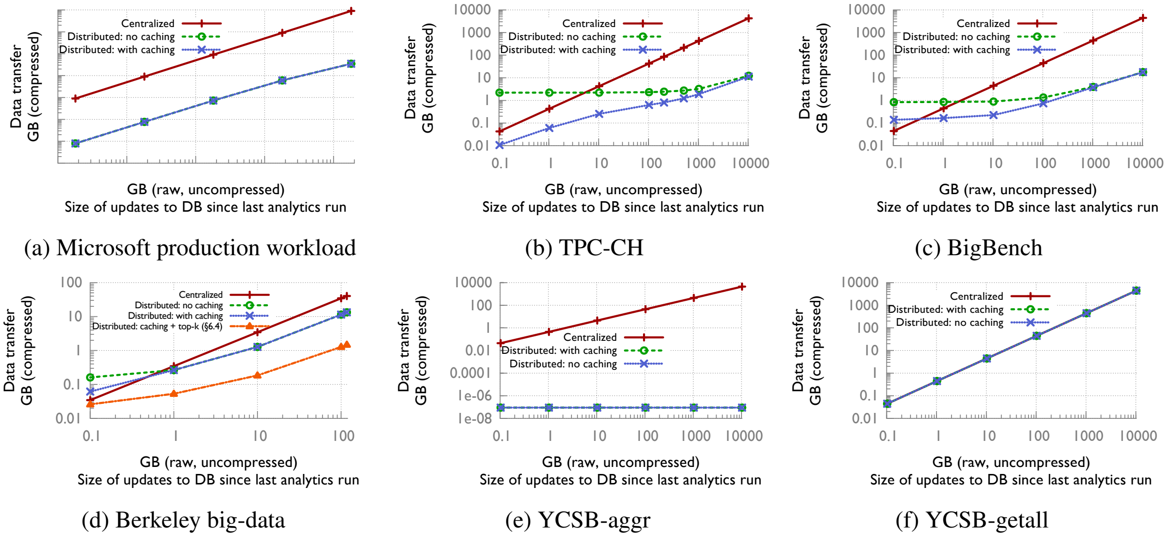
Extensibility

Geode is designed to support arbitrary application domains; as such the core of the system does not include optimizations for specific kinds of queries. However, the system is an extensible substrate on top of which users can easily implement narrow optimizations targeted at their needs. To demonstrate the flexibility of our system we implemented two function-specific optimizations: an exact algorithm for top-k queries [7], originally proposed in a CDN analytics setting and recently used by JetStream [34]; and an approximate percentile algorithm from the sensor networks literature [36]. We evaluate the benefit from these optimizations in §6.4.

6 Experimental Evaluation

We now investigate the following questions experimentally: How much of a bandwidth savings does our system actually yield on real workloads at multi-terabyte scales (§6.1)? What is the runtime overhead of collecting the (pseudo-distributed) measurements needed by our optimizer (§6.2)? What is the tradeoff between solution quality and processing time in the optimizer (§6.3)? Can implementing narrow application-specific optimizations yield significant further bandwidth cost reduction (§6.4)?

[3] This simple strategy is sufficient because Hive does not support correlated subqueries.



(a) Microsoft production workload

(b) TPC-CH

(c) BigBench

(d) Berkeley big-data

(e) YCSB-aggr

(f) YCSB-getall

Figure 8: End-to-end evaluation of all six workloads

6.1 Large-scale evaluation

We ran experiments measuring Geode performance on a range of workloads, on two Geode deployments: a distributed deployment across three data centers in the US, Europe and Asia, and a large centralized cluster on which we simulated a multi-data center setup. Specifically, we ran experiments on both deployments up to the 25 GB scale (and validated that the results were identical), but used the centralized cluster exclusively for all experiments on a Microsoft production workload and all experiments larger than 25 GB on other workloads. This was because running experiments at the multi-terabyte scale we evaluate would have otherwise cost tens of thousands of dollars in bandwidth in a fully distributed deployment.

We tested six workloads.

Microsoft production workload: This use case consists of a monitoring infrastructure collecting tens of TBs of service health/telemetry data daily at geographically distributed data centers. The data are continuously replicated to a central location and analyzed using Hive. The bulk of the load comes from a few tens of canned queries run every day producing aggregate reports on service utilization and infrastructure health.

TPC-CH: The TPC-CH benchmark [9] by Cole et al. models the database for a large-scale product retailer such as Amazon, and is a joint OLTP + OLAP benchmark constructed by combining the well-known TPC-C OLTP benchmark and the TPC-H OLAP benchmark.

BigBench: BigBench [18] is a recently proposed benchmark for big-data systems modeling a large scale product retailer that sells items online and in-store, collecting various information from customers (including reviews and click logs) in the process. Analytics consists of a core of Hive queries along with some non-relational machine learning operations that further process the relational output. We do not implement the non-relational component explicitly, but model it as a black box analyst interacting with Geode — Geode's task is to compute the results the non-relational black box needs as input.

Big-data: The big-data benchmark [6], developed by the AMPLab at UC Berkeley, models a database generated from HTTP server logs. The analytical queries in this benchmark are parametric: each has a single parameter that can be adjusted to tune the volume of data transfer that would be required to process it. In our experiments we set the normalized value ($\in [0,1]$) of each parameter to 0.5, to make each query require median data transfer.

YCSB-aggr, YCSB-getall: We defined these two very simple benchmarks to demonstrate the best- and worst-case scenarios for our system, respectively. Both benchmarks operate using the YCSB [11] database and OLTP workload, configured with database schema:

```
    table(key, field1, field2)
```

The OLTP workload is constituted by transactions that add a single row with `field1` a randomly chosen digit in the range $[0,9]$ and `field2` a random 64-bit integer. The difference between the two benchmarks is solely in their analytical workload.

YCSB-aggr's analytical workload consists of the single query `SELECT field1, AVG(field2) FROM Table GROUP BY field1`. Since there are only 10 distinct values of `field1`, Geode achieves significant aggregation, requiring only 10 rows (partial sum and count for each distinct `field1`) from each data center.

YCSB-getall's analytical workload is a single query asking for every row in the table (`SELECT * FROM Table`). Here no WABD solution can do better than centralized analytics.

We evaluate all six workloads by measuring the data transfer needed for both centralized and distributed execution for varying volumes of changes to the base data in between runs of the analytical workload. Our workload optimizer consistently picks among the best of the centralized and distributed solutions at each point, so that Geode's performance would be represented by the min of all the graphs in each plot. We omit the min line to avoid crowding the figures.

Figure 8 shows results for all six workloads. (We are required to obfuscate the scale of the axes of Figure 8a due to the proprietary nature of the underlying data.) We note a few key observations.

In general, the centralized approach performs relatively better when update rates are low, actually outperforming distributed execution at very low rates in 2 of the 6 workloads. This is because low volumes mean frequent analytics running on mostly unchanged data. Distributed execution performs better at higher update rates.

Caching significantly improves performance at low update rates in TPC-CH, BigBench and Berkeley bigdata: for instance, performance with caching always outperforms centralized execution in the TPC-CH benchmark, while performance without caching is worse for volumes < 6 GB per OLAP run. However, at high update rates, caching is ineffective since redundancy in the query answers is minimal. Caching does not help in the YCSB workloads because small changes to the base data end up changing analytics results completely in both benchmarks, and in the Microsoft production workload because every query tagged all output rows with a query execution timestamp, which interacts poorly with the row-based approach we use to compute deltas (more sophisticated diffs can overcome this limitation).

At the largest scales we tested, distributed execution outperformed the centralized approach by $150-360\times$ in four of our six workloads (YCSB-aggr, Microsoft prod., TPC-CH, and BigBench). The improvement was only $3\times$ in the Big-Data with normal distributed execution, but when we implemented the special optimization for top-k queries [7] we discussed in §5, the improvement went up to $27\times$ — we discuss details in §6.4. Finally, YCSB-getall was deliberately designed so that distributed execution could not outperform the centralized approach, and we find that this is indeed the case.

6.2 Optimizer: Runtime overhead

The pseudo-distributed execution method we use to collect data transfer measurements can slow down query execution (although it never worsens bandwidth cost, as we discussed in §4.2). We measured the added overhead for all the queries we tested in §6.1.

In all our workloads, we found that the latency overhead compared to normal distributed Geode was contained in the $<$**20%** range. Given the scale-out nature of the Hive backend, this is easily compensated for by increasing parallelism. Note also that this overhead is only occasionally felt, since in our architecture the optimizer operates on a much slower timescale than normal query execution. E.g. if queries are run once a day and the optimizer runs once a month, pseudo-distributed execution only affects $1/30 = 3.3\%$ of the query runs.

Further, this overhead could be reduced in many cases by using separate lightweight statistics-gathering queries to estimate transfers, instead of full-fledged pseudo-distributed runs. For instance, for the query in Figure 2, we could instead run a `SELECT sum( len(pageURL) + len (pageRank)) FROM PageInfo WHERE ...` query to estimate the size of the join, and then determine the size of the final output by executing the query using a normal (as opposed to a pseudo-distributed) join.

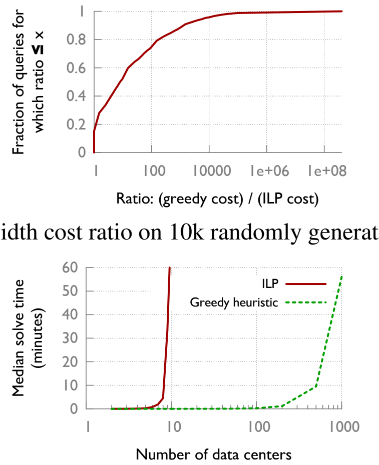
6.3 Optimizer Performance, Running time

The optimizer consists of two components, the Calcite++ centralized SQL query planner, and a site selection + data replication solver. Calcite++ is responsible for a very small proportion of the optimizer's running time, completing in < 10 s for all the queries in §6.1. The majority of the time spent by the optimizer is in the site selection and data replication phase, for which we defined two solutions: a slower but optimal integer linear program, and a faster but potentially suboptimal greedy heuristic (§4.3). We now investigate the relative performance of these two approaches.

We first compare the optimality gap between the two solutions by evaluating their performance on: (i) the real workloads from §6.1, and (ii) simulations on randomly generated SQL workloads.

In all the workloads we tested in §6.1, the optimality gap is small. The greedy strategy performs remarkably well, identifying the same solution as the ILP in over 98% of the queries we tested. It does fail in some instances, however. For example, the BigBench [18] benchmark has a query which joins a sales log table with itself to identify pairs of items that are frequently ordered together. The heuristic greedily pushes the join down to each data center, resulting in a large list of item pairs stored partitioned across several data centers. But it is then forced to retrieve the entire list to a single data center in order to compute the final aggregate. By contrast, the ILP correctly detects that copying the entire order log to a single data center first would be much cheaper.

In order to compare the strategies' performance in a more general setting, we simulated their performance on randomly generated SQL queries. We generated 10,000 random chain-join queries of the form `SELECT * FROM` T_1 `JOIN` T_2 `... JOIN` T_k `USING(col)`, where each table has schema T_i`(col INT)`, k chosen randomly



(a) Bandwidth cost ratio on 10k randomly generated queries

(b) Running time for workloads of the same scale as §6.1

Figure 9: ILP and greedy heuristic comparison

between 2 and 10. In each query we chose table sizes and join selectivities according to a statistical model by Swami and Gupta [37], which tries to cover a large range of realistic query patterns, generating e.g. both queries which heavily aggregate the input they consume in each step, as well as queries which "expand" their inputs.

Figure 9a shows the results we obtained. The greedy heuristic and the ILP identified the same strategy in around 16% of the queries. In the remaining 84% the ILP performs better: 8× better in the median, and more than 8 orders of magnitude better in the tail. The worst performance generally arises when the heuristic compounds multiple errors of the kind described in the example above. The results show that the gap between the true optimum and the greedy strategy can be substantial.

However, this optimality gap turns out to be difficult to bridge at large scales. Figure 9b shows the running times of both approaches for workloads of the same size as the largest in §6.1. The ILP's running time grows very quickly, taking more than an hour with just 10 data centers. By contrast, the greedy heuristic takes less than a minute even at the 100 data centers scale, although as we have seen this can come at the expense of a loss in solution quality.

We are actively evaluating a hybrid strategy that first uses the greedy heuristic to generate an initial solution, and then uses the ILP for a best-effort search for better alternatives starting from the initial greedy solution, until a specified running time bound. We defer reporting results until a thorough evaluation.

We note again that many of the results reported in this section were based on simulating synthetic workloads, albeit ones that were designed to be realistic [37]. The question of how well both approaches will perform on practical workloads (beyond those in §6.1, where we saw that the greedy heuristic was competitive) remains open, and can only be answered in the future, as analytical workloads rise in sophistication to take advantage of the cost reduction achieved by geo-distributed execution.

6.4 Function-specific optimizations

We close by showing how performance could be improved even further by leveraging optimizations targeted at specific classes of queries from past work. We evaluate the two optimizations we discussed in §5: for top-k queries [7] and for approximate percentile queries [36].

Both algorithms proved quite effective on applicable queries. The top-k algorithm directly benefited the most data-intensive query in the Berkeley big-data benchmark (Figure 8d), achieving a further 9× reduction in data transfer over normal distributed execution. And in a sales-value percentile query we defined on the TPC-CH benchmark database, the approximate percentile algorithm achieved170× less data transfer than exact computation with $< 5\%$ error, and 30× less with $< 1\%$ error.

There is a vast range of optimizations from several related fields one can leverage in the WABD setting — Geode serves as a convenient framework on which these optimizations can be layered.

7 Limitations and Open Problems

Several considerations arise when designing a global analytics framework. We chose to focus solely on minimizing bandwidth costs, while handling fault-tolerance requirements and respecting sovereignty constraints. We did not attempt to address:

Latency. Our focus was entirely on reducing data transfer volume and large scale, and we made no attempt to optimize analytics query latency. It is likely that in many cases the problems of minimizing bandwidth usage and minimizing latency coincide, but effort charactacterizing the differences is necessary.

Consistency. We support a relaxed eventual consistency consistency model. This suffices for many use cases which only care about aggregates and trends, but the problem of building a WABD solution for applications requiring stronger consistency guarantees remains open.

Privacy. Geode addresses regulatory restrictions by limiting where base data can be copied. However, we allow arbitrary queries on top of base data, and make no attempt to proscribe data movement by queries. While this suffices for scenarios where all queries are carefully vetted before they are allowed to execute, an automated solution, which would necessitate a differential privacy [15] or privacy-preserving computation [26] mechanism, would be interesting to pursue.

Other bandwidth cost models. We assumed each network link has a constant \$/byte cost. Supporting other cost models, such as ones based on 95th %ile bandwidth usage, would require modifying the workload optimizer.

Other data models. We have concentrated on WABD for a relational model, but similar issues of bandwidth minimization and latency/regulatory constraints arise in other data models as well, such as Map-Reduce or even computational models that go beyond querying such as machine learning. The fundamental issues, limited bandwidth and the choice between various levels of distributed and centralized computing, remain the same. We discuss these challenges further in [43].

8 Related Work

Unlike parallel databases running in a single LAN [13, 20], where latencies are assumed to be uniform and low, we have non-uniform latency and wide-area bandwidth costs. Work on distributed databases and view maintenance, starting as early as [8, 5] and surveyed in [27, 33], handles efficient execution of arbitrary queries assuming a fixed data partitioning and placement. By contrast, we are able to assume a slowly evolving workload that the system can be optimized for (§4), and automatically replicate data for performance and fault-tolerance while handling regulatory constraints. The focus on analytics instead of transactions, the much larger scale of WABD, and the focus on bandwidth as a measure further differentiates WABD from distributed databases [33].

Spanner [12] focuses on consistency and low-latency transaction support, and is not designed to optimize analytics costs. A complete solution would complement Spanner-like consistent transactions with cost-efficient analytics as in Geode. The Mesa [22] data warehouse geo-replicates data for fault tolerance, as we do, but continues to process analytical queries within a single data center. Stream-processing databases [23, 34] process long-standing continuous queries, transforming a dispersed collection of input streams into an output stream. The significant focus in this area has been on relatively simple data models with data always produced at the edge, with (typically degraded) summaries transmitted to the center, in contrast with the relational model we consider.

Jetstream [34] is an example of stream processing for data structured as OLAP cubes that focuses, as we do, on bandwidth as a metric; however, its data model is not as rich as a relational model. Joins, for example, are not allowed. Further, the sytem relies enitirely on aggregation and approximation to reduce bandwidth, techniques that are not sufficient for the analytical queries we focus on.

Sensor networks [31] share our assumption of limited network bandwidth, but not our large scale or the breadth of our computational model. However, some sensor network techniques can be of interest in WABD: for instance, the approximate percentile algorithm we tested in §6.4 was originally proposed for a sensor network.

Hive [39], Pig [32], Spark [45] and similar systems can provide analytics on continuously updated data, but to the best of our knowledge have not been tested in multi-data center deployments (and are certainly not optimized for this scenario). PNUTS/Sherpa[10] does support geograhically distributed partitions but lays out data to optimize latency (by moving a "master" copy close to where it is commonly used) and not to minimize analytics cost.

Volley [2] addresses placement issues for data while accounting for wide-area bandwidth and user latencies, but without our additional constraint of handling rich analytics. RACS [1] distributes a key-vaue store, not a database, across data centers and focuses on fault-tolerance, not bandwidth. Distributed file systems share our assumption of limited bandwidth, and the caching mechanism we use can be viewed as operating on cached files of answers to earlier analytical queries. However, distributed file systems do not share our relational data model or the query planning problem we face.

PigOut [25], developed concurrently with our work, supports Pig [32] queries on data partitioned across data centers, but targets a simpler two-step computational model than ours and focuses on optimizing individual queries in isolation.

In a recent paper [43] we discussed the vision of geo-distributed analytics for a more general computational model with DAGs of tasks. This was a vision paper that focused on non-SQL models and did not include the detailed description or evaluation of our techniques we present here.

9 Conclusion

Current data volumes and heuristics such as data reduction allow centralizing analytics to barely suffice in the short term, but the approach will soon be rendered untenable by rapid growth in data volumes relative to network capacity and rising regulatory interest in proscribing data movement. In this paper we proposed an alternative: Wide-Area Big Data. Our Hive-based prototype, Geode, achieves up to a 360× bandwidth reduction at multi-TB scales compared to centralization on both production workloads and standard benchmarks. Our approach revisits the classical database problem of query planning from a networking perspective, both in terms of constraints such as bandwidth limits and autonomous policies, as well as solutions such as sub-query deltas and pseudo-distributed execution.

Acknowledgements

We would like to thank our shepherd Hakim Weatherspoon and the anonymous reviewers for their valuable suggestions. We gratefully acknowledge the support of an Alfred P. Sloan Research Fellowship.

References

[1] H. Abu-Libdeh, L. Princehouse, and H. Weatherspoon. RACS: a case for cloud storage diversity. In *SoCC 2010*.

[2] S. Agarwal, J. Dunagan, N. Jain, S. Saroiu, A. Wolman, and H. Bhogan. Volley: Automated data placement for geo-distributed cloud services. In *NSDI 2010*.

[3] A. Auradkar, C. Botev, S. Das, D. D. Maagd, A. Feinberg, P. Ganti, L. Gao, B. Ghosh, K. Gopalakrishna, B. Harris, J. Koshy, K. Krawez, J. Kreps, S. Lu, S. Nagaraj, N. Narkhede, S. Pachev, I. Perisic, L. Qiao, T. Quiggle, J. Rao, B. Schulman, A. Sebastian, O. Seeliger, A. Silberstein, B. Shkolnik, C. Soman, R. Sumbaly, K. Surlaker, S. Topiwala, C. Tran, B. Varadarajan, J. Westerman, Z. White, D. Zhang, and J. Zhang. Data infrastructure at LinkedIn. *2014 IEEE 30th International Conference on Data Engineering*, 0:1370–1381, 2012.

[4] C. Ballinger. Born to be parallel: Why parallel origins give Teradata database an enduring performance edge. `http://www.teradata.com/white-papers/born-to-be-parallel-teradata-database`.

[5] P. A. Bernstein, N. Goodman, E. Wong, C. L. Reeve, and J. B. Rothnie, Jr. Query processing in a system for distributed databases (SDD-1). *ACM Transactions on Database Systems*, 1981.

[6] The Big Data Benchmark. `https://amplab.cs.berkeley.edu/benchmark/`.

[7] P. Cao and Z. Wang. Efficient top-k query calculation in distributed networks. In *PODC '04*, PODC '04.

[8] W. W. Chu and P. Hurley. Optimal query processing for distributed database systems. *IEEE Trans. Comput.*

[9] R. Cole, F. Funke, L. Giakoumakis, W. Guy, A. Kemper, S. Krompass, H. Kuno, R. Nambiar, T. Neumann, M. Poess, K.-U. Sattler, M. Seibold, E. Simon, and F. Waas. The mixed workload ch-benchmark. In *DBTest '11*, DBTest '11, pages 8:1–8:6, New York, NY, USA, 2011. ACM.

[10] B. F. Cooper, R. Ramakrishnan, U. Srivastava, A. Silberstein, P. Bohannon, H.-A. Jacobsen, N. Puz, D. Weaver, and R. Yerneni. PNUTS: Yahoo!'s hosted data serving platform. *VLDB*, 2008.

[11] B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears. Benchmarking cloud serving systems with ycsb. In *SoCC '10*.

[12] J. C. Corbett, J. Dean, M. Epstein, A. Fikes, C. Frost, J. Furman, S. Ghemawat, A. Gubarev, C. Heiser, P. Hochschild, W. Hsieh, S. Kanthak, E. Kogan, H. Li, A. Lloyd, S. Melnik, D. Mwaura, D. Nagle, S. Quinlan, R. Rao, L. Rolig, Y. Saito, M. Szymaniak, C. Taylor, R. Wang, and D. Woodford. Spanner: Google's globally-distributed database. In *OSDI 2012*.

[13] D. DeWitt and J. Gray. Parallel database systems: the future of high performance database systems. *Commun. ACM*.

[14] A. Dey. Yahoo cross data-center data movement. `http://yhoo.it/1nPRImNl`, 2010.

[15] C. Dwork. Differential privacy: A survey of results. In M. Agrawal, D. Du, Z. Duan, and A. Li, editors, *Theory and Applications of Models of Computation*, volume 4978 of *Lecture Notes in Computer Science*, pages 1–19. Springer Berlin Heidelberg, 2008.

[16] European Commission press release. Commission to pursue role as honest broker in future global negotiations on internet governance. `http://europa.eu/rapid/press-release_IP-14-142_en.htm`.

[17] L. Gao. On inferring autonomous system relationships in the internet. *IEEE/ACM Trans. Netw.*, 9(6):733–745, Dec. 2001.

[18] A. Ghazal, T. Rabl, M. Hu, F. Raab, M. Poess, A. Crolotte, and H.-A. Jacobsen. Bigbench: Towards an industry standard benchmark for big data analytics. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*, SIGMOD '13, pages 1197–1208, New York, NY, USA, 2013. ACM.

[19] K. Glerum, K. Kinshumann, S. Greenberg, G. Aul, V. Orgovan, G. Nichols, D. Grant, G. Loihle, and G. Hunt. Debugging in the (very) large: Ten years of implementation and experience. In *Proceedings of the ACM SIGOPS 22Nd Symposium on Operating Systems Principles*, SOSP '09, pages 103–116, New York, NY, USA, 2009. ACM.

[20] G. Graefe. Query evaluation techniques for large databases. *ACM Comput. Surv.*

[21] A. Greenberg, J. Hamilton, D. A. Maltz, and P. Patel. The cost of a cloud: Research problems in data center networks. *ACM CCR*, 39(1), 2008.

[22] A. Gupta, F. Yang, J. Govig, A. Kirsch, K. Chan, K. Lai, S. Wu, S. Dhoot, A. Kumar, A. Agiwal, S. Bhansali, M. Hong, J. Cameron, M. Siddiqi, D. Jones, J. Shute, A. Gubarev, S. Venkataraman, and D. Agrawal. Mesa: Geo-replicated, near real-time, scalable data warehousing. In *VLDB*, 2014.

[23] J.-H. Hwang, Y. Xing, U. Cetintemel, and S. Zdonik. A cooperative, self-configuring high-availability solution for stream processing. In *Data Engg. Workshop 2007*.

[24] T. Inc. Global Internet Geography. `http://www.telegeography.com/research-services/global-internet-geography/`, 2014.

[25] K. Jeon, S. Chandrashekhara, F. Shen, S. Mehra, O. Kennedy, and S. Y. Ko. Pigout: Making multiple hadoop clusters work together. In *Big Data, 2014 IEEE International Conference on*, pages 100–109, Oct 2014.

[26] F. Kerschbaum. Privacy-preserving computation. In B. Preneel and D. Ikonomou, editors, *Privacy Technologies and Policy*, volume 8319 of *Lecture Notes in Computer Science*, pages 41–54. Springer Berlin Heidelberg, 2014.

[27] D. Kossmann. The state of the art in distributed query processing. *ACM Computing Surveys*, 32(4):422–469, Dec. 2000.

[28] N. Laoutaris, M. Sirivianos, X. Yang, and P. Rodriguez. Inter-datacenter bulk transfers with netstitcher. In *SIGCOMM 2011*.

[29] G. Lee, J. Lin, C. Liu, A. Lorek, and D. Ryaboy. The unified logging infrastructure for data analytics at Twitter. *PVLDB*, 2012.

[30] G. Lohman. Is query optimization a "solved" problem? `http://wp.sigmod.org/?p=1075`, April 2014.

[31] S. Madden. Database abstractions for managing sensor network data. *Proc. of the IEEE*, 98(11):1879–1886, 2010.

[32] C. Olston, B. Reed, U. Srivastava, R. Kumar, and A. Tomkins. Pig Latin: a not-so-foreign language for data processing. In *SIGMOD 2008*.

[33] M. T. Özsu and P. Valduriez. *Principles of distributed database systems*. Springer, 2011.

[34] A. Rabkin, M. Arye, S. Sen, V. S. Pai, and M. J. Freedman. Aggregation and degradation in JetStream: Streaming analytics in the wide area. In *NSDI 2014*.

[35] M. Rost and K. Bock. Privacy by design and the new protection goals. *DuD, January*, 2011.

[36] N. Shrivastava, C. Buragohain, D. Agrawal, and S. Suri. Medians and beyond: New aggregation techniques for sensor networks. In *Proceedings of the 2Nd International Conference on Embedded Networked Sensor Systems*, SenSys '04, pages 239–249, New York, NY, USA, 2004. ACM.

[37] A. Swami and A. Gupta. Optimization of large join queries. In *Proceedings of the 1988 ACM SIGMOD International Conference on Management of Data*, SIGMOD '88, pages 8–17, New York, NY, USA, 1988. ACM.

[38] A. L. Tatarowicz, C. Curino, E. P. C. Jones, and S. Madden. Lookup tables: Fine-grained partitioning for distributed databases. In *Proceedings of the 2012 IEEE 28th International Conference on Data Engineering*, ICDE '12, pages 102–113, Washington, DC, USA, 2012. IEEE Computer Society.

[39] A. Thusoo, J. S. Sarma, N. Jain, Z. Shao, P. Chakka, N. Zhang, S. Antony, H. Liu, and R. Murthy. Hive - a petabyte scale data warehouse using hadoop. In *ICDE 2010*.

[40] A. Thusoo, Z. Shao, S. Anthony, D. Borthakur, N. Jain, J. Sen Sarma, R. Murthy, and H. Liu. Data warehousing and analytics infrastructure at Facebook. SIGMOD, 2010.

[41] A. Tridgell and P. Mackerras. The rsync algorithm, 1996.

[42] M. Vrable, S. Savage, and G. M. Voelker. Cumulus: Filesystem backup to the cloud. *Trans. Storage*, 5(4):14:1–14:28, Dec. 2009.

[43] A. Vulimiri, C. Curino, P. B. Godfrey, K. Karanasos, and G. Varghese. WANalytics: Analytics for a Geo-Distributed Data-Intensive World. In *Conference on Innovative Data Systems Research (CIDR 2015)*, January 2015.

[44] J. Wiener and N. Boston. Facebook's top open data problems. `https://research.facebook.com/blog/1522692927972019/facebook-s-top-open-data-problems/`, 2014.

[45] M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauly, M. J. Franklin, S. Shenker, and I. Stoica. Resilient distributed datasets: a fault-tolerant abstraction for in-memory cluster computing. In *NSDI 2012*.

Succinct: Enabling Queries on Compressed Data

Rachit Agarwal
UC Berkeley

Anurag Khandelwal
UC Berkeley

Ion Stoica
UC Berkeley

Abstract

Succinct is a data store that enables efficient queries directly on a compressed representation *of the input data*. Succinct uses a compression technique that allows random access into the input, thus enabling efficient storage and retrieval of data. In addition, Succinct natively supports a wide range of queries including count and search of arbitrary strings, range and wildcard queries. What differentiates Succinct from previous techniques is that Succinct supports these queries *without* storing indexes — all the required information is embedded within the compressed representation.

Evaluation on real-world datasets show that Succinct requires an order of magnitude lower memory than systems with similar functionality. Succinct thus pushes more data in memory, and provides low query latency for a larger range of input sizes than existing systems.

1 Introduction

High-performance data stores, *e.g.* document stores [1, 6], key-value stores [5,9,23,24,26,38,39,43] and multi-attribute NoSQL stores [3, 19, 21, 25, 35, 48], are the bedrock of modern cloud services. While existing data stores provide efficient abstractions for storing and retrieving data using primary keys, interactive queries on values (or, secondary attributes) remains a challenge.

To support queries on secondary attributes, existing data stores can use two main techniques. At one extreme, systems such as column oriented stores, simply scan the data [10, 36]. However, data scans incur high latency for large data sizes, and have limited throughput since queries typically touch all machines[1]. At the other extreme, one can construct indexes on queried attributes [3, 6, 35]. When stored in-memory, these indexes are not only fast, but can achieve high throughput since it is possible to execute each query on a single machine. The main disadvantage of indexes is their high memory footprint. Evaluation of popular open-source data stores [6,35] using real-world datasets (§6) shows that indexes can be as much as 8× larger than the input data size. Traditional compression techniques can reduce the memory footprint but suffer from degraded throughput since data needs to be decompressed even for simple queries. Thus, existing data stores either resort to using complex memory management techniques for identifying and caching "hot" data [5, 6, 26, 35] or simply executing queries off-disk or off-SSD [25]. In either case, latency and throughput advantages of indexes drop compared to in-memory query execution.

We present Succinct, a distributed data store that operates at a new point in the design space: memory efficiency close to data scans and latency close to indexes. Succinct queries on secondary attributes, however, touch all machines; thus, Succinct may achieve lower throughput than indexes when the latter fits in memory. However, due to its low memory footprint, Succinct is able to store more data in memory, avoiding latency and throughput degradation due to off-disk or off-SSD query execution for a much larger range of input sizes than systems that use indexes.

Succinct achieves the above using two key ideas. First, Succinct stores *an entropy-compressed representation of the input data* that allows random access, enabling efficient storage and retrieval of data. Succinct's data representation natively supports count, search, range and wildcard queries *without* storing indexes — all the required information is embedded within this compressed representation. Second, Succinct *executes queries directly on the compressed representation*, avoiding data scans and decompression. What makes Succinct a unique system is that it not only stores a compressed representation of the input data, but also provides functionality similar to systems that use indexes along with input data.

Specifically, Succinct makes three contributions:

- Enables efficient queries directly on a compressed representation of the input data. Succinct achieves this using (1) a new data structure, in addition to adapting data structures from theory literature [32, 44–46], to compress the input data; and (2) a new query algorithm that executes random access, count,

[1] Most data stores shard data by rows, and one needs to scan all rows. Even if data is sharded by columns, one needs to touch multiple machines to construct the row(s) in the query result.

search, range and wildcard queries directly on the compressed representation (§3). In addition, Succinct provides applications the flexibility to tradeoff memory for faster queries and vice versa (§4).

- Efficiently supports data appends by chaining multiple stores, each making a different tradeoff between write, query and memory efficiency (§4): (1) a small log-structured store optimized for fine-grained appends; (2) an intermediate store optimized for query efficiency while supporting bulk appends; and (3) an immutable store that stores most of the data, and optimizes memory using Succinct's data representation.
- Exposes a minimal, yet powerful, API that operates on flat unstructured files (§2). Using this simple API, we have implemented many powerful abstractions for semi-structured data on top of Succinct including document store (*e.g.*, MongoDB [6]), key-value store (*e.g.*, Dynamo [23]), and multi-attribute NoSQL store (*e.g.*, Cassandra [35]), enabling efficient queries on both primary and secondary attributes.

We evaluate Succinct against MongoDB [6], Cassandra [35], HyperDex [25] and DB-X, an industrial columnar store that supports queries via data scans. Evaluation results show that Succinct requires 10−11× lower memory than data stores that use indexes, while providing similar or stronger functionality. In comparison to traditional compression techniques, Succinct's data representation achieves lower decompression throughput but supports point queries directly on the compressed representation. By pushing more data in memory and by executing queries directly on the compressed representation, Succinct achieves dramatically lower latency and higher throughput (sometimes an order of magnitude or more) compared to above systems even for moderate size datasets.

2 Succinct Interface

Succinct exposes a simple interface for storing, retrieving and querying flat (unstructured) files; see Figure 1. We show in §2.1 that this simple interface already allows us to model many powerful abstractions including MongoDB [6], Cassandra [35] and BigTable [19], enabling efficient queries on semi-structured data.

The application submits and compresses a flat `file` using `compress`; once compressed, it can invoke a set of powerful primitives directly on the compressed file. In particular, the application can append new data using `append`, can perform random access using `extract` that returns an uncompressed buffer starting at an arbitrary offset in original `file`, and count number of occurrences of any arbitrary string using `count`.

```
f = compress(file)
append(f, buffer)
buffer = extract(f, offset, len)
cnt = count(f, str)
[offset1, ...] = search(f, str)
[offset1, ...] = rangesearch(f, str1, str2)
[[offset1, len1], ...]
  = wildcardsearch(f, prefix, suffix, dist)
```

Figure 1: Interface exposed by Succinct (see §2).

Arguably, the most powerful operation provided by Succinct is `search` which takes as an argument an *arbitrary* string (*i.e.*, not necessarily word-based) and returns offsets of all occurrences in the uncompressed `file`. For example, if `file` contains `abbcdeabczabgz`, invoking `search(f, "ab")` will return offsets `[0, 6, 10]`. While `search` returns an array of offsets, we provide a convenient iterator interface in our implementation. What makes Succinct unique is that `search` not only runs on the *compressed* representation but is also efficient, that is, does not require scanning the `file`.

Succinct provides two other search functions, again on arbitrary input strings. First, `rangesearch` returns the offsets of all strings between `str1` and `str2` in lexicographical order. Second, `wildcardsearch(f, prefix, suffix, dist)` returns an array of tuples. A tuple contains the offset and the length of a string with the given `prefix` and `suffix`, and whose distance between the prefix and suffix does not exceed `dist`, measured in number of input characters. Suppose again that file `f` contains `abbcdeabczabgz`, then `wildcardsearch(f, "ab", "z", 2)` will return tuples `[6, 9]` for `abcz`, and `[10, 13]` for `abgz`. Note that we do not return the tuple corresponding to `abbcdeabcz` as the distance between the prefix and suffix of this string is greater than 2.

2.1 Extensions for semi-structured data

Consider a logical collection of records of the form `(key, avpList)`, where `key` is a unique identifier, and `avpList` is a list of attribute value pairs, i.e., `avpList = ((attrName1, value1),... (attrNameN, valueN))`. To enable queries using Succinct API, we encode `avpList` within Succinct data representation; see Figure 2. Specifically, we transform the semi-structured data into a flat file with each attribute value separated by a delimiter unique to that attribute. In addition, Succinct internally stores a mapping from each attribute to the corresponding delimiter, and a mapping from `key` to `offset` into the flat file where corresponding `avpList` is encoded.

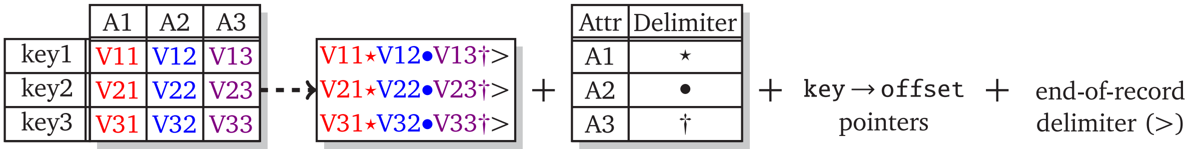


Figure 2: Succinct supports queries on semi-structured data by transforming the input data into flat files (see §2.1).

Succinct executes `get` queries using `extract` API along with the `key→offset` pointers, and `put` queries using the `append` API. The `delete` queries are executed lazily, similar to [8, 10], using one explicit bit per record which is set upon record deletion; subsequent queries ignore records with set bit. Applications can also query individual attributes; for instance, search for string `val` along attribute `A2` is executed as `search(val•)` using the Succinct API, and returns every `key` whose associated attribute `A2` value matches `val`.

Flexible schema, record sizes and data types. Succinct, by mapping semi-structured data into a flat file and by using delimiters, does not impose any restriction on `avpList`. Indeed, Succinct supports single-attribute records (*e.g.*, Dynamo [23]), multiple-attribute records (*e.g.*, BigTable [19]), and even a collection of records with varying number of attributes. Moreover, using its key → offset pointers, Succinct supports the realistic case of records varying from a few bytes to a few kilobytes [17]. Succinct currently supports primitive data types (`strings, integers, floats`), and can be extended to support a variety of data structures and data types including composite types (`arrays, lists, sets`). See [16] for a detailed discussion.

3 Querying on Compressed Data

We describe the core techniques used in Succinct. We briefly recall techniques from theory literature that Succinct uses, followed by Succinct's entropy-compressed representation (§3.1) and a new algorithm that operates directly on the compressed representation (§3.2).

Existing techniques. Classical search techniques are usually based on tries or suffix trees [13, 47]. While fast, even their *optimized representations* can require 10–20× more memory than the input size [33, 34]. Burrows-Wheeler Transform (BWT) [18] and Suffix arrays [12, 40] are two memory efficient alternatives, but still require 5× more memory than the input size [33]. FM-indexes [27–30] and Compressed Suffix Arrays [31, 32, 44–46] use compressed representation of BWT and suffix arrays, respectively, to further reduce the space requirement. Succinct adapts compressed suffix arrays due to their simplicity and relatively better performance for large datasets. We describe the basic idea behind Compressed Suffix Arrays.

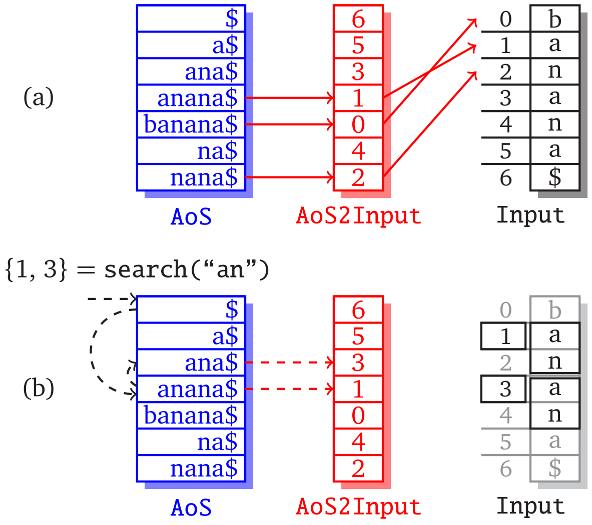


Figure 3: An example for input file banana$. AoS stores suffixes in the input in lexicographically sorted order. (a) AoS2Input maps each suffix in AoS to its location in the input (solid arrows). (b) Illustration of search using AoS and AoS2Input (dashed arrows). Suffixes being sorted, AoS allows binary search to find the smallest AoS index whose suffix starts with searched string (in this case "an"); the largest such index is found using another binary search. The result on the original input is showed on the right to aid illustration.

Let Array of Suffixes (AoS) be an array containing all suffixes in the input file in lexicographically sorted order. AoS along with two other arrays, AoS2Input and Input2AoS[2], is sufficient to implement the search and the random access functionality without storing the input file. This is illustrated in Figure 3 and Figure 4.

Note that for a file with n characters, AoS has size $O(n^2)$ bits, while AoS2Input and Input2AoS have size $n\lceil \log n \rceil$ bits since the latter two store integers in range 0 to $n-1$. The space for AoS, AoS2Input and Input2AoS is reduced by storing only a subset of values; the remaining values are computed on the fly using a set of pointers, stored in NextCharIdx array, as illustrated in Figure 5, Figure 6 and Figure 7, respectively.

[2] AoS2Input and Input2AoS, in this paper, are used as convenient names for Suffix array and Inverse Suffix Array, respectively.

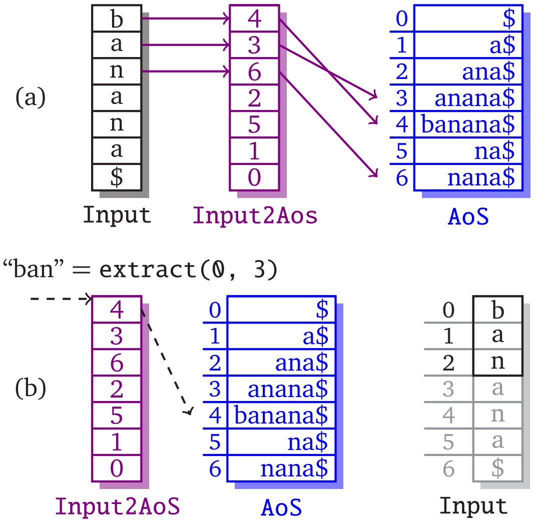


Figure 4: (a) The Input2AoS provides the *inverse mapping* of AoS2Input, from each index in the input to the index of the corresponding suffix in AoS (solid arrows). (b) Illustration of extract using AoS and Input2AoS (dashed arrows). The result on the original input is showed on the right to aid illustration.

The NextCharIdx array is compressed using a two-dimensional representation; see Figure 8. Specifically, the NextCharIdx values in each column of the two-dimensional representation constitute an *increasing sequence* of integers[3]. Each column can hence be independently compressed using delta encoding [2, 7, 11].

3.1 Succinct data representation

Succinct uses the above data representation with three main differences. We give a high-level description of these differences; see [16] for a detailed discussion.

First, Succinct uses a more space-efficient representation of AoS2Input and Input2AoS by using a sampling by "value" strategy. In particular, for sampling rate α, rather than storing values at "indexes" $\{0, \alpha, 2\alpha, \dots\}$ as in Figure 6 and Figure 7, Succinct stores all AoS2Input values that are a multiple of α. This allows storing each sampled value `val` as `val`/α, leading to a more space-efficient representation. Using $\alpha = 2$ for example of Figure 6, for instance, the sampled AoS2Input values are `{6,0,4,2}`, which can be stored as `{3,0,2,1}`. Sampled Input2AoS then becomes `{1,3,2,0}` with i-th value being the index into sampled AoS2Input where i is stored. Succinct stores a small amount of additional information to locate sampled AoS2Input indexes.

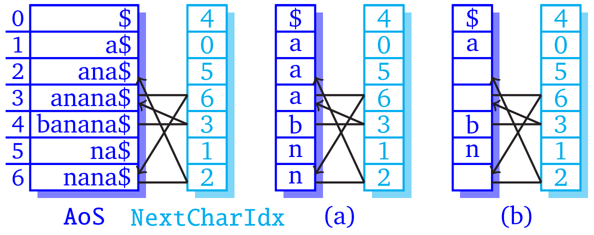


Figure 5: Reducing the space usage of AoS: NextCharIdx stores pointers from each suffix S to the suffix S' after removing the first character from S. (a) for each suffix in AoS, only the first character is stored. NextCharIdx pointers allow one to reconstruct suffix at any AoS index. For instance, starting from AoS[4] and following pointers, we get the original AoS entry "banana$". (b) Since suffixes are sorted, only the first AoS index at which each character occurs (*e.g.*, `{($,0),(a,1),(b,4),(n,5)}`) need be stored; a binary search can be used to locate character at any index.

Second, Succinct achieves a more space-efficient representation for NextCharIdx using the fact that values in each row of the two-dimensional representation constitute a *contiguous sequence* of integers[4]. Succinct uses its own *Skewed Wavelet Tree* data structure, based on Wavelet Trees [32, 44], to compress each row independently. Skewed Wavelet Trees allow looking up NextCharIdx value at any index without any decompression. The data structure and lookup algorithm are described in detail in [16]. These ideas allow Succinct to achieve 1.25–3× more space-efficient representation compared to existing techniques [7, 11, 31].

Finally, for semi-structured data, Succinct supports dictionary encoding along each attribute to further reduce the memory footprint. This is essentially orthogonal to Succinct's own compression; in particular, Succinct dictionary encodes the data along each attribute before constructing its own data structures.

3.2 Queries on compressed data

Succinct executes queries directly on the compressed representation from §3.1. We describe the query algorithm assuming access to uncompressed data structures; as discussed earlier, any value not stored in the compressed representation can be computed on the fly.

Succinct executes an `extract` query as illustrated in Figure 7 on Input2AoS representation from §3.1. A strawman algorithm for `search` would be to perform two binary searches as in Figure 3. However, this algorithm suffers from two inefficiencies. First, it executes binary searches on the entire AoS2Input array; and sec-

[3] Proof: Consider two suffixes `cX<cY` in a column (indexed by character "`c`"). By definition, NextCharIdx values corresponding to `cX` and `cY` store AoS indexes corresponding to suffixes `X` and `Y`. Since `cX<cY` implies `X<Y` and since AoS stores suffixes in sorted order, `NextCharIdx[cX]<NextCharIdx[cY]`; hence the proof.

[4] Intuitively, any row indexed by `rowID` contains NextCharIdx values that are pointers into suffixes starting with the string `rowID`; since suffixes are sorted, these must be contiguous set of integers.

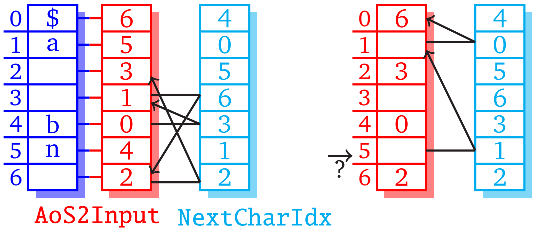


Figure 6: Reducing the space usage of AoS2Input. (left) Since AoS2Input stores locations of suffixes in AoS, NextCharIdx maps AoS2Input values to next larger value. That is, NextCharIdx[idx] stores the AoS2Input index that stores AoS2Input[idx]+1[5]; (right) only a few sampled values need be stored; unsampled values can be computed on the fly. For instance, starting AoS2Input[5] and following pointers twice, we get the next larger sampled value 6. Since each pointer increases value by 1, the desired value is $6-2=4$.

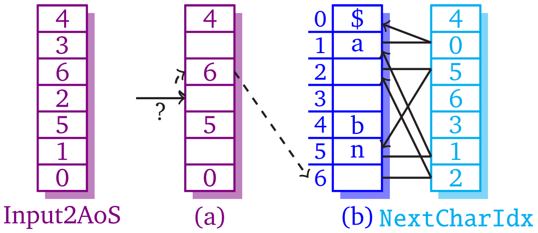


Figure 7: Reducing the space usage of Input2AoS. (a) only a few sampled values need be stored; (b) `extract` functionality of Figure 4 is achieved using sampled values and NextCharIdx. For instance, to execute `extract(3, 3)`, we find the next smaller sampled index (Input2AoS[2]) and corresponding suffix (AoS[2]="nana$"). We then remove the first character since the difference between the desired index and the closest sampled index was 1; hence the result "ana$".

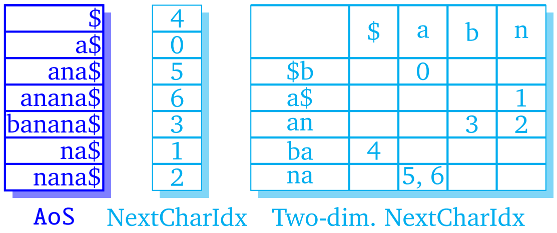


Figure 8: Two-dimensional NextCharIdx representation. Columns are indexed by all unique characters and rows are indexed by all unique t−length strings in input file, both in sorted order. A value belongs to a cell `(colID, rowID)` if corresponding suffix has `colID` as first character and `rowID` as following t characters. For instance, `NextCharIdx[3]=5` and `NextCharIdx[4]=6` are contained in cell `(a, na)`, since both start with "`a`" and have "`na`" as following two characters.

ond, each step of the binary search requires computing the suffix at corresponding AoS index for comparison purposes. Succinct uses a query algorithm that overcomes these inefficiencies by aggressively exploiting the two-dimensional NextCharIdx representation.

Recall that the cell `(colID, rowID)` in two-dimensional NextCharIdx representation corresponds to suffixes that have `colID` as the first character and `rowID` as the following t characters. Succinct uses this to perform binary search in cells rather than the entire AoS2Input array. For instance, consider the query `search("anan")`; all occurrences of string "`nan`" are contained in the cell ⟨`n,an`⟩. To find all occurrences of string `anan`, our algorithm performs a binary search only in the cell ⟨`a,na`⟩ in the next step. Intuitively, after this step, the algorithm has the indexes for which suffixes start with "`a`" and are followed by "`nan`", the desired string. For a string of length m, the above algorithm performs $2(m-t-1)$ binary searches, two per NextCharIdx cell [16], which is far more efficient than executing two binary searches along the entire AoS2Input array for practical values of m. In addition, the algorithm does not require computing any of the AoS suffixes during the binary searches. For a 16GB file, Succinct's query algorithm achieves a 2.3× speed-up on an average and 19× speed-up in the best case compared to the strawman algorithm.

Range and Wildcard Queries. Succinct implements `rangesearch` and `wildcardsearch` using the `search` algorithm. To implement `rangesearch(f, str1, str2)`, we find the smallest AoS index whose suffix starts with string `str1` and and the largest AoS index whose suffix starts with string `str2`. Since suffixes are sorted, the returned range of indices necessarily contain all strings that are lexicographically contained between `str1` and `str2`. To implement `wildcardsearch(f, prefix, suffix, dist)`, we first find the offsets of all prefix and suffix occurrences, and return all possible combinations such that the difference between the suffix and prefix offsets is positive and no larger than `dist` (after accounting for the prefix length).

4 Succinct Multi-store Design

Succinct incorporates its core techniques into a write-friendly multi-store design that chains multiple individual stores each making a different tradeoff between write, query and memory efficiency. This section describes the design and implementation of the individual stores and their synthesis to build Succinct.

[5] Proof: Let S be a suffix and S' be the suffix after removing first character from S. If S starts at location `loc`, then S' starts at `loc+1`. NextCharIdx stores pointers from S to S'. Since AoS2Input stores locations of suffixes in input, NextCharIdx maps value `loc` in AoS2Input to AoS2Input index that stores the next larger value (`loc+1`).

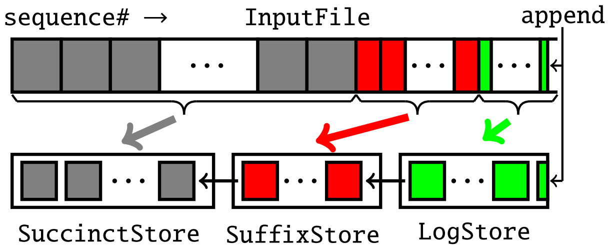


Figure 9: Succinct uses a write-optimized LogStore that supports fine-grained appends, a query-optimized SuffixStore that supports bulk appends, and a memory-optimized SuccinctStore. New data is appended to the end of LogStore. The entire data in LogStore and SuffixStore constitutes a single partition of SuccinctStore. The properties of each of the stores are summarized in Table 1.

Table 1: Properties of individual stores. Data size estimated for 1TB original uncompressed data on a 10 machine 64GB RAM cluster. Memory estimated based on evaluation (§6).

	Succinct Store	**Suffix Store**	**Log Store**
Stores	Comp. Data (§3.1)	Data + AoS2Input	Data + Inv. Index
Appends	-	Bulk	Fine
Queries	§3.2	Index	Scans+ Inv. Index
#Machines	$n-2$	1	1
%Data(est.)	$> 99.98\%$	$< 0.016\%$	$< 0.001\%$
Memory	$\approx 0.4\times$	$\approx 5\times$	$\approx 9\times$

Succinct design overview. Succinct chains *three* individual stores as shown in Figure 9; Table 1 summarizes the properties of the individual stores. New data is appended into a write-optimized *LogStore*, that executes queries via in-memory data scans; the queries are further sped up using an inverted index that supports fast fine-grained updates. An intermediate store, *SuffixStore*, supports bulk appends and aggregates larger amounts of data before compression is initiated. Scans at this scale are simply inefficient. SuffixStore thus supports fast queries using *uncompressed* data structures from §3; techniques in place ensure that these data structures do not need to be updated upon bulk appends. SuffixStore raw data is periodically transformed into an immutable entropy-compressed store *SuccinctStore* that supports queries directly on the compressed representation. The average memory footprint of Succinct remains low since most of data is contained in the memory-optimized SuccinctStore.

4.1 LogStore

LogStore is a write-optimized store that executes data `append` via main memory writes, and other queries via data scans. Memory efficiency is not a goal for LogStore since it contains a small fraction of entire dataset.

One choice for LogStore design is to let cores concurrently execute read and write requests on a single shared partition and exploit parallelism by assigning each query to one of the cores. However, concurrent writes scale poorly and require complex techniques for data structure integrity [39,41,42]. Succinct uses an alternative design, partitioning LogStore data into multiple partitions, each containing a small amount of data. However, straightforward partitioning may lead to incorrect results if the query searches for a string that spans two partitionsLogStore thus uses *overlapping partitions*, each annotated with the starting and the ending offset corresponding to the data "owned" by the partition. The overlap size can be configured to expected string search length (default is 1MB). New data is always appended to the most recent partition.

LogStore executes an `extract` request by reading the data starting at the offset specified in the request. While this is fast, executing `search` via data scans can still be slow, requiring tens of milliseconds even for 250MB partition sizes. Succinct avoids scanning the entire partition using an "inverted index" per partition that supports fast updates. This index maps short length (default is three character) strings to their locations in the partition; queries then need to scan characters starting only at these locations. The index is memory inefficient, requiring roughly 8× the size of LogStore data, but has little affect on Succinct's average memory since LogStore itself contains a small fraction of the entire data. The speed-up is significant allowing Succinct to scan, in practice, up to 1GB of data within a millisecond. The index supports fast updates since, upon each write, only locations of short strings in the new data need to be appended to corresponding entries in the index.

4.2 SuffixStore

SuffixStore is an intermediate store between LogStore and entropy-compressed SuccinctStore that serves two goals. First, to achieve good compression, SuffixStore accumulates and queries much more data than LogStore before initiating compression. Second, to ensure that LogStore size remains small, SuffixStore supports bulk data appends without updating any existing data.

Unfortunately, LogStore approach of fast data scans with support of inverted index does not scale to data sizes in SuffixStore due to high memory footprint and data scan latency. SuffixStore thus stores *uncompressed* AoS2Input array (§3) and executes search queries via binary search (Figure 3). SuffixStore avoids storing AoS by storing the original data that allows random access for comparison during binary search, as well as, for extract queries; these queries are fast since AoS2Input is uncompressed. SuffixStore achieves the second goal using excessive partitioning, with overlapping partitions similar to LogStore. Bulk appends from LogStore are executed at partition granularity, with the entire LogStore data constituting a single partition of SuffixStore. AoS2Input is constructed per partition to ensure that bulk appends do not require updating any existing data.

4.3 SuccinctStore

SuccinctStore is an immutable store that contains most of the data, and is thus designed for memory efficiency. SuccinctStore uses the entropy-compressed representation from §3.1 and executes queries directly on the compressed representation as described in §3.2. SuccinctStore's design had to resolve two additional challenges.

First, Succinct's memory footprint and query latency depends on multiple tunable parameters (*e.g.*, AoS2Input and Input2AoS sampling rate and string lengths for indexing NextCharIdx rows). While default parameters in SuccinctStore are chosen to operate on a sweet spot between memory and latency, Succinct will lose its advantages if input data is too large to fit in memory even after compression using default parameters. Second, LogStore being extremely small and SuffixStore being latency-optimized makes SuccinctStore a latency bottleneck. Hence, Succinct performance may deteriorate for workloads that are skewed towards particular SuccinctStore partitions.

Succinct resolves both these challenges by enabling applications to tradeoff memory for query latency. Specifically, Succinct enables applications to select AoS2Input and Input2AoS sampling rate; by storing fewer sampled values, lower memory footprint can be achieved at the cost of higher latency (and vice versa). This resolves the first challenge above by reducing the memory footprint of Succinct to avoid answering queries off-disk[6]. This also helps resolving the second challenge by increasing the memory footprint of overloaded partitions, thus disproportionately speeding up these partitions for skewed workloads.

We discuss data transformation from LogStore to SuffixStore and from SuffixStore to SuccinctStore in §5.

5 Succinct Implementation

We have implemented three Succinct prototypes along with extensions for semi-structured data (§2.1) — in Java running atop Tachyon [37], in Scala running atop Spark [51], and in C++. We discuss implementation details of the C++ prototype that uses roughly 5,200 lines of code. The high-level architecture of our Succinct prototype is shown in Figure 10. The system consists of a central coordinator and a set of storage servers, one server each for LogStore and SuffixStore, and the remaining servers for SuccinctStore. All servers share a similar architecture modulo the differences in the storage format and query execution, as described in §3.

The coordinator performs two tasks. The first task is *membership management*, which includes maintaining a list of active servers in the system by having each server send periodic heartbeats. The second task is *data management*, which includes maintaining an up-to-date collection of pointers to quickly locate the desired data during query execution. Specifically, the coordinator maintains two set of pointers: one that maps file offsets to partitions that contain the data corresponding to the offsets, and the other one that maps partitions to machines that store those partitions. As discussed in §2.1, an additional set of key → offset pointers are also maintained for supporting queries on semi-structured data.

Clients connect to one of the servers via a light-weight *Query Handler* (QH) interface; the same interface is also used by the server to connect to the coordinator and to other servers in the system. Upon receiving a query from a client, the QH parses the query and identifies whether the query needs to be forwarded to a single server (for `extract` and `append` queries) or to all the other servers (for `count` and `search` queries).

In the case of an `extract` or `append` query, QH needs to identify the server to which the query needs to be forwarded. One way to do this is to forward the query to the coordinator, which can then lookup its sets of pointers and forward the query to the appropriate server. However, this leads to the coordinator becoming a bottleneck. To avoid this, the pointers are cached at each server. Since the number of pointers scales only in the number of partitions and servers, this has minimal impact on Succinct's memory footprint. The coordinator ensures that pointer updates are immediately pushed to each of the servers. Using these pointers, an `extract` query is redirected to the QH of the appropriate machine, which then locates the appropriate partition and extracts the desired data.

In the case of a `search` query, the QH that receives the query from the client forwards the query to all the

[6] Empirically, Succinct can achieve a memory footprint comparable to GZip. When even the GZip-compressed data does not fit in memory, the only option for any system is to answer queries off disk.

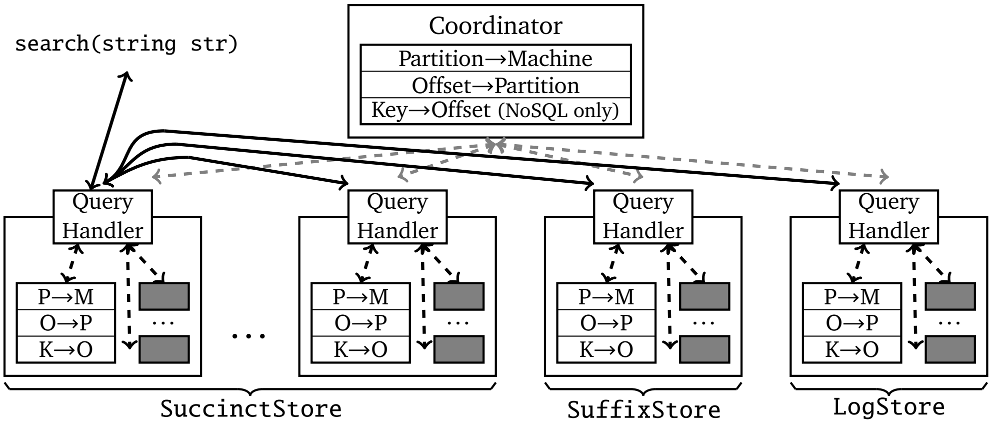


Figure 10: Succinct system architecture. Server and coordinator functionalities are described in §5. Each server uses a light-weight Query Handler interface to (1) interact with coordinator; (2) redirect queries to appropriate partitions and/or servers; and (3) local and global result aggregation. P→M, O→P and K→O are the same pointers as stored at the coordinator.

other QHs in the system. In turn, each QH runs multiple tasks to search all local partitions in parallel, then aggregates the results, and sends these results back to the initiator, that is, to the QH that initiated the query (see Figure 10). Finally, the initiator returns the aggregated result to the client. While redirecting queries using QHs reduces the coordinator load, QHs connecting to all other QHs may raise some scalability concerns. However, as discussed earlier, due to its efficient use of memory, Succinct requires many fewer servers than other in-memory data stores, which helps scalability.

Data transformation between stores. LogStore aggregates data across multiple partitions before transforming it into a single SuffixStore partition. LogStore is neither memory nor latency constrained; we expect each LogStore partition to be smaller than 250MB even for clusters of machines with 128GB RAM. Thus, AoS2Input for LogStore data can be constructed at LogStore server itself, using an efficient linear-time, linear-memory algorithm [50]. Transforming SuffixStore data into a SuccinctStore partition requires a merge sort of AoS2Input for each of the SuffixStore partitions, scanning the merged array once to construct Input2AoS and NextCharIdx, sampling AoS2Input and Input2AoS, and finally compressing each row of NextCharIdx. Succinct could use a single over-provisioned server for SuffixStore to perform this transformation at the SuffixStore server itself but currently does this in the background.

Failure tolerance and recovery. The current Succinct prototype requires manually handling: (1) coordinator failure; (2) data failure and recovery; and (3) adding new servers to an existing cluster. Succinct could use traditional solutions for maintaining multiple coordinator replicas with a consistent view. Data failure and recovery can be achieved using standard replication-based techniques. Finally, since each SuccinctStore contains multiple partitions, adding a new server simply requires moving some partitions from existing servers to the new server and updating pointers at servers. We leave incorporation of these techniques and evaluation of associated overheads to future work.

6 Evaluation

We now perform an end-to-end evaluation of Succinct's memory footprint (§6.1), throughput (§6.2) and latency (§6.3).

Compared systems. We evaluate Succinct using the NoSQL interface extension (§2.1), since it requires strictly more space and operations than the unstructured file interface. We compare Succinct against several open-source and industrial systems that support search queries: MongoDB [6] and Cassandra [35] using secondary indexes; HyperDex [25] using hyperspace hashing; and an industrial columnar-store DB-X, using in-memory data scans[7].

We configured each of the system for no-failure scenario. For HyperDex, we use the dimensionality as recommended in [25]. For MongoDB and Cassandra, we used the most memory-efficient indexes. These indexes do not support substring searches and wildcard

[7] For HyperDex, we encountered a previously known bug [4] that crashes the system during query execution when inter-machine latencies are highly variable. For DB-X, distributed experiments require access to the industrial version. To that end, we only perform microbenchmarks for HyperDex and DB-X for Workloads A and C.

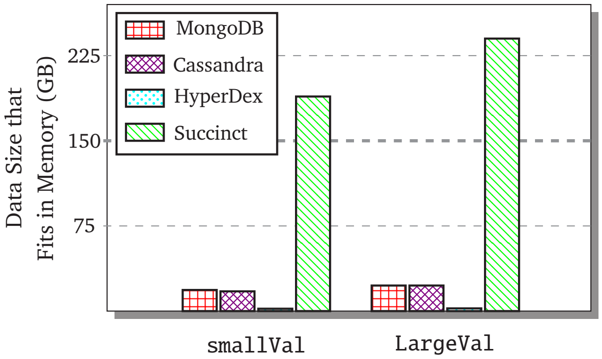


Figure 11: Input data size that each system fits in-memory on a distributed cluster with 150GB main memory (thick horizontal line). Succinct pushes 10-11× larger amount of data in memory compared to popular open-source data stores, while providing similar or stronger functionality.

searches. HyperDex and DB-X do not support wildcard searches. Thus, the evaluated systems provide slightly weaker functionality than Succinct. Finally, for Succinct, we disabled dictionary encoding to evaluate the performance of Succinct techniques in isolation.

Datasets, Workloads and Cluster. We use two multi-attribute record datasets, one `smallVal` and one `largeVal` from Conviva customers as shown in Table 2. The workloads used in our evaluation are also summarized in Table 2. Our workloads closely follow YCSB workloads; in particular, we used YCSB to generate query keys and corresponding query frequencies, which were then mapped to the queries in our datasets (for each of read, write, and search queries). All our experiments were performed on Amazon EC2 m1.xlarge machines with 15GB RAM and 4 cores, except for DB-X where we used pre-installed r2.2xlarge instances. Each of the system was warmed up for 5 minutes to maximize the amount of data cached in available memory.

6.1 Memory Footprint

Figure 11 shows the amount of input data (without indexes) that each system fits across a distributed cluster with 150GB main memory. Succinct supports in-memory queries on data sizes larger than the system RAM; note that Succinct results do *not* use dictionary encoding and also include pointers required for NoSQL interface extensions (§2.1, §5). MongoDB and Cassandra fit roughly 10–11× less data than Succinct due to storing secondary indexes along with the input data. HyperDex not only stores large metadata but also avoids touching multiple machines by storing a copy of the entire record with each subspace, thus fitting up to 126× less data than Succinct.

6.2 Throughput

We now evaluate system throughput using a distributed 10 machine Amazon EC2 cluster. Figure 12 shows throughput results for `smallVal` and `LargeVal` datasets across the four workloads from Table 2.

Workload A. When MongoDB and Cassandra can fit datasets in memory (17GB for `smallVal` and 23GB for `LargeVal` across a 150GB RAM cluster), Succinct's relative performance depends on record size. For small record sizes, Succinct achieves higher throughput than MongoDB and Cassandra. For MongoDB, the routing server becomes a throughput bottleneck; for Cassandra, the throughput is lower because more queries are executed off-disk. However, when record sizes are large, Succinct achieves slightly lower throughput than MongoDB due to increase in Succinct's `extract` latency.

When MongoDB and Cassandra data does not fit in memory, Succinct achieves better throughput since it performs in-memory operations while MongoDB and Cassandra have to execute some queries off-disk. Moreover, we observe that Succinct achieves consistent performance across data sizes varying from tens of GB to hundreds of GB.

Workload B. MongoDB and Succinct observe reduced throughput when a small fraction of queries are append queries. MongoDB throughput reduces since indexes need to be updated upon each write; for Succinct, LogStore writes become a throughput bottleneck. Cassandra being write-optimized observes minimal reduction in throughput. We observe again that, as we increase the data sizes from 17GB to 192GB (for `SmallVal`) and from 23GB to 242GB (for `LargeVal`), Succinct's throughput remains essentially unchanged.

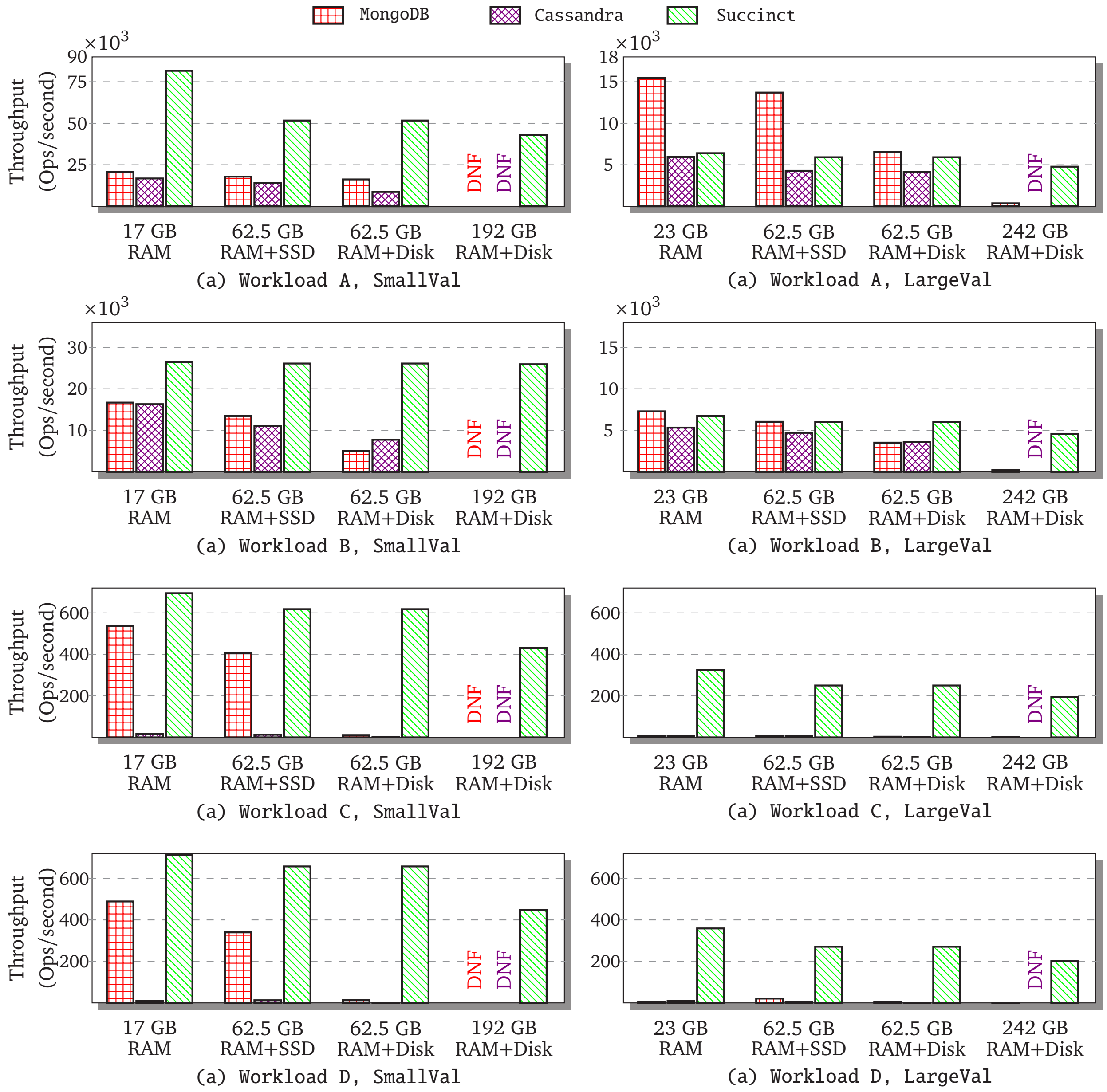
Workload C. For search workloads, we expect MongoDB and Cassandra to achieve high throughput due to storing indexes. However, Cassandra requires scanning indexes for search queries leading to low throughput. The case of MongoDB is more interesting. For datasets with fewer number of attributes (`SmallVal` dataset), MongoDB achieves high throughput due to caching being more effective; for `LargeVal` dataset, MongoDB search throughput reduces significantly even when the entire index fits in memory. When MongoDB indexes do not fit in memory, Succinct achieves 13–134× higher throughput since queries are executed in-memory.

As earlier, even with 10× increase in data size (for both `smallVal` and `LargeVal`), Succinct throughput reduces minimally. As a result, Succinct's performance for large datasets is comparable to the performance of MongoDB and Cassandra for much smaller datasets.

Table 2: (left) Datasets used in our evaluation; (right) Workloads used in our evaluation. All workloads use a query popularity that follows a Zipf distribution with skewness 0.99, similar to YCSB [20].

	Size (Bytes)		#Attr-ibutes	#Records (Millions)
	Key	Value		
smallVal	8	≈ 140	15	123–1393
LargeVal	8	≈ 1300	98	19–200

	Workload	Remarks
A	100% Reads	YCSB workload C
B	95% Reads, 5% appends	YCSB workload D
C	100% Search	-
D	95% Search, 5% appends	YCSB workload E



(a) Workload A, SmallVal

(a) Workload A, LargeVal

(a) Workload B, SmallVal

(a) Workload B, LargeVal

(a) Workload C, SmallVal

(a) Workload C, LargeVal

(a) Workload D, SmallVal

(a) Workload D, LargeVal

Figure 12: Succinct throughput against MongoDB and Cassandra for varying datasets, data sizes and workloads. MongoDB and Cassandra fit 17GB of SmallVal dataset and 23GB of LargeVal dataset in memory; Succinct fits 192GB and 242GB, respectively. DNF denote the experiment did not finish after 100 hours of data loading, mostly due to index construction time. Note that top four figures have different y-scales.

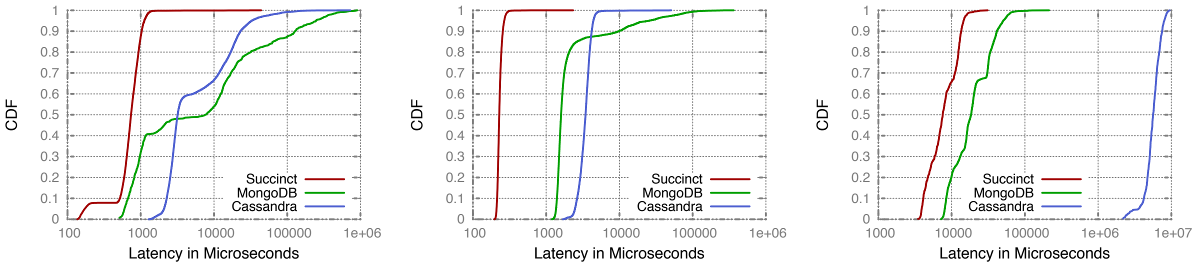


Figure 13: Succinct's latency for `get` (left), `put` (center) and `search` (right) against MongoDB and Cassandra for `smallVal` dataset when data and index fits in memory (best case for MongoDB and Cassandra). Discussion in §6.3.

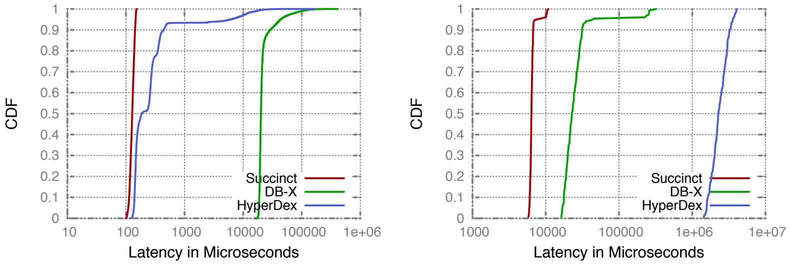


Figure 14: Succinct's latency for `get` (left) and `search` (right) against HyperDex and DB-X for `smallVal` 10GB dataset on a single machine. HyperDex uses subspace hashing and DB-X uses in-memory data scans for search. Discussion in §6.3.

Workload D. The search throughput for MongoDB and Cassandra becomes even worse as we introduce 5% appends, precisely due to the fact that indexes need to be updated upon each append. Unlike Workload B, Succinct search throughput does not reduce with appends, since writes are no more a bottleneck. As earlier, Succinct's throughput scales well with data size.

Note that the above discussion holds even when MongoDB and Cassandra use SSDs to store the data that does not fit in memory. When such is the case, throughput reduction is lower compared to the case when data is stored on disk; nevertheless, the trends remain unchanged. Specifically, Succinct is able to achieve better or comparable performance than SSD based systems for a much larger range of input values.

6.3 Latency

We now compare Succinct's latency against two sets of systems: (1) systems that use indexes to support queries (MongoDB and Cassandra) on a distributed 10 node Amazon EC2 cluster; and (2) systems that perform data scans along with metadata to support queries (HyperDex and DB-X) using a single-machine system. To maintain consistency across all latency experiments, we only evaluate cases where all systems (except for HyperDex) fit the entire data in memory.

Succinct against Indexes. Figure 13 shows that Succinct achieves comparable or better latency than MongoDB and Cassandra even when all data fits in memory. Indeed, Succinct's latency will get worse if record sizes are larger. For writes, we note that both MongoDB and Cassandra need to update indexes upon each write, leading to higher latency. For search, MongoDB achieves good latency since MongoDB performs a binary search over an in-memory index, which is similar in complexity to Succinct's search algorithm. Cassandra requires high latencies for search queries due to much less efficient utilization of available memory.

Succinct against data scans. Succinct's latency against systems that do not store indexes is compared in Figure 14. HyperDex achieves comparable latency for `get` queries; `search` latencies are higher since due to its high memory footprint, HyperDex is forced to answer most queries off-disk. DB-X being a columnar store is not optimized for `get` queries, thus leading to high latencies. For `search` queries, DB-X despite optimized in-memory data scans is around 10× slower at high percentiles because data scans are inherently slow.

6.4 Throughput versus Latency

Figure 15 shows the throughput versus latency results for Succinct, for both `get` and `search` queries

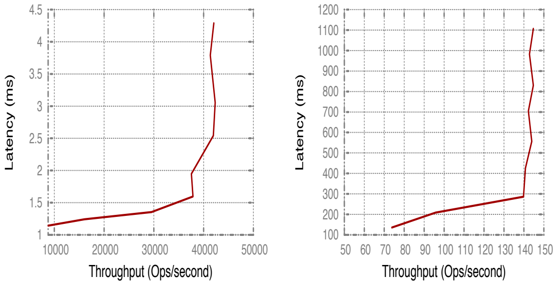


Figure 15: Throughput versus latency for Succinct, for `get` (left) and for `search` (right).

for a fully loaded 10 machine cluster with `smallVal` 192GB dataset. The plot shows that Succinct latency and throughput results above are for the case of a fully loaded system.

6.5 Sequential Throughput

Our evaluation results for workload A and B used records of sizes at most 1300bytes per query. We now discuss Succinct's performance in terms of throughput for long sequential reads. We ran a simple microbenchmark to evaluate the performance of Succinct over a single `extract` request for varying sizes of reads. Succinct achieves a constant throughput of 13Mbps using a single core single thread implementation, irrespective of the read size; the throughput increases linearly with number of threads and/or cores. This is essentially a tradeoff that Succinct makes for achieving high throughput for short reads and for search queries using a small memory footprint. For applications that require large number of sequential reads, Succinct can overcome this limitation by keeping the original uncompressed data to support sequential reads, of course at the cost of halving the amount of data that Succinct pushes into main memory. The results from Figure 11 show that Succinct will still push 5-5.5× more data than popular open-source systems with similar functionality.

7 Related Work

Succinct's goals are related to three key research areas:

Queries using secondary indexes. To support point queries, many existing data stores store indexes/metadata [3, 6, 25, 35] in addition to the original data. While indexes achieve low latency and high throughput when they fit in memory, their performance deteriorates significantly when queries are executed off-disk. Succinct requires more than 10× lower memory than systems that store indexes, thus achieving higher throughput and lower latency for a much larger range of input sizes than systems that store indexes.

Queries using data scans. Point queries can also be supported using data scans. These are memory efficient but suffer from low latency and throughput for large data sizes. Most related to Succinct is this space are columnar stores [10, 15, 22, 36, 49]. The most advanced of these [10] execute queries either by scanning data or by decompressing the data on the fly (if data compressed [14]). As shown in §6, Succinct achieves better latency and throughput by avoiding expensive data scans and decompression.

Theory techniques. Compressed indexes has been an active area of research in theoretical computer science since late 90s [27–30, 32, 44–46]. Succinct adapts data structures from above works, but improves both the memory and the latency by using new techniques (§3). Succinct further resolves several challenges to realize these techniques into a practical data store: (1) efficiently handling updates using a multi-store design; (2) achieving better scalability by carefully exploiting parallelism within and across machines; and (3) enabling queries on semi-structured data by encoding the structure within a flat file.

8 Conclusion

In this paper, we have presented Succinct, a distributed data store that supports a wide range of queries while operating at a new point in the design space between data scans (memory-efficient, but high latency and low throughput) and indexes (memory-inefficient, low latency, high throughput). Succinct achieves memory footprint close to that of data scans by storing the input data in an entropy-compressed representation that supports random access, as well as a wide range of analytical queries. When indexes fit in memory, Succinct achieves comparable latency, but lower throughput. However, due to its low memory footprint, Succinct is able to store more data in memory, avoiding latency and throughput reduction due to off-disk or off-SSD query execution for a much larger range of input sizes than systems that use indexes.

Acknowledgments

This research is supported in part by NSF CISE Expeditions Award CCF-1139158, LBNL Award 7076018, and DARPA XData Award FA8750-12-2-0331, and gifts from Amazon Web Services, Google, SAP, The Thomas and Stacey Siebel Foundation, Adatao, Adobe, Apple, Inc., Blue Goji, Bosch, C3Energy, Cisco, Cray, Cloudera, EMC, Ericsson, Facebook, Guavus, Huawei, Informatica, Intel, Microsoft, NetApp, Pivotal, Samsung, Splunk, Virdata and VMware.

References

[1] CouchDB. http://couchdb.apache.org.

[2] Delta Encoding. http://en.wikipedia.org/wiki/Delta_encoding.

[3] Elasticsearch. http://www.elasticsearch.org.

[4] Hyperdex Bug. https://groups.google.com/forum/#!msg/hyperdex-discuss/PUIpjMPEiAI/I3ZImpU7OtkJ.

[5] MemCached. http://www.memcached.org.

[6] MongoDB. http://www.mongodb.org.

[7] Pizza&Chili Corpus: Compressed Indexes and their Testbeds. http://pizzachili.dcc.uchile.cl/indexes/Compressed_Suffix_Array/.

[8] Presto. http://prestodb.io.

[9] Redis. http://www.redis.io.

[10] SAP HANA. http://www.saphana.com/.

[11] SDSL. https://github.com/simongog/sdsl-lite.

[12] Suffix Array. http://en.wikipedia.org/wiki/Suffix_array.

[13] Suffix Tree. http://en.wikipedia.org/wiki/Suffix_tree.

[14] Vertica Does Not Compute on Compressed Data. http://tinyurl.com/l36w8xs.

[15] D. J. Abadi, S. R. Madden, and M. Ferreira. Integrating Compression and Execution in Column-Oriented Database Systems. In *ACM International Conference on Management of Data (SIGMOD)*, 2006.

[16] R. Agarwal, A. Khandelwal, and I. Stoica. Succinct: Enabling Queries on Compressed Data. In *Technical Report, UC Berkeley*, 2014.

[17] B. Atikoglu, Y. Xu, E. Frachtenberg, S. Jiang, and M. Paleczny. Workload Analysis of a Large-scale Key-value Store. In *ACM SIGMETRICS Performance Evaluation Review*, volume 40, pages 53–64, 2012.

[18] M. Burrows and D. J. Wheeler. A block-sorting lossless data compression algorithm. 1994.

[19] F. Chang, J. Dean, S. Ghemawat, W. C. Hsieh, D. A. Wallach, M. Burrows, T. Chandra, A. Fikes, and R. E. Gruber. Bigtable: A Distributed Storage System for Structured Data. In *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2006.

[20] B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears. Benchmarking Cloud Serving Systems with YCSB. In *ACM Symposium on Cloud Computing (SoCC)*, 2010.

[21] J. C. Corbett, J. Dean, M. Epstein, A. Fikes, C. Frost, J. Furman, S. Ghemawat, A. Gubarev, C. Heiser, P. Hochschild, et al. Spanner: Google's Globally-distributed Database. In *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2012.

[22] Daniel J. Abadi and Samuel R. Madden and Nabil Hachem. Column-Stores vs. Row-Stores: How Different Are They Really? In *ACM International Conference on Management of Data (SIGMOD)*, 2008.

[23] G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, and W. Vogels. Dynamo: Amazon's Highly Available Key-value Store. In *ACM Symposium on Operating Systems Principles (SOSP)*, 2007.

[24] A. Dragojević, D. Narayanan, O. Hodson, and M. Castro. FaRM: Fast Remote Memory. In *USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2014.

[25] R. Escriva, B. Wong, and E. G. Sirer. HyperDex: A Distributed, Searchable Key-value Store. In *ACM Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication (SIGCOMM)*, 2012.

[26] B. Fan, D. G. Andersen, and M. Kaminsky. MemC3: Compact and Concurrent MemCache with Dumber Caching and Smarter Hashing. In *USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2013.

[27] P. Ferragina and G. Manzini. Opportunistic Data Structures with Applications. In *IEEE Symposium on Foundations of Computer Science (FOCS)*, 2000.

[28] P. Ferragina and G. Manzini. An Experimental Study of a Compressed Index. *Information Sciences*, 135(1):13–28, 2001.

[29] P. Ferragina and G. Manzini. An Experimental Study of an Opportunistic Index. In *ACM-SIAM Symposium on Discrete Algorithms (SODA)*, 2001.

[30] P. Ferragina and G. Manzini. Indexing Compressed Text. *Journal of the ACM (JACM)*, 52(4):552–581, 2005.

[31] R. Grossi, A. Gupta, and J. S. Vitter. High-order Entropy-compressed Text Indexes. In *ACM-SIAM Symposium on Discrete Algorithms (SODA)*, 2003.

[32] R. Grossi and J. S. Vitter. Compressed Suffix Arrays and Suffix Trees with Applications to Text Indexing and String Matching. *SIAM Journal on Computing*, 35(2):378–407, 2005.

[33] W.-K. Hon, T. W. Lam, W.-K. Sung, W.-L. Tse, C.-K. Wong, and S.-M. Yiu. Practical aspects of Compressed Suffix Arrays and FM-Index in Searching DNA Sequences. In *Workshop on Algorithm Engineering and Experiments and the First Workshop on Analytic Algorithmics and Combinatorics (ALENEX/ANALC)*, 2004.

[34] S. Kurtz. Reducing the Space Requirement of Suffix Trees. *Software: Practice and Experience*, 29(13):1149–1171, 1999.

[35] A. Lakshman and P. Malik. Cassandra: A Decentralized Structured Storage System. *ACM SIGOPS Operating Systems Review*, 44(2):35–40, 2010.

[36] A. Lamb, M. Fuller, R. Varadarajan, N. Tran, B. Vandiver, L. Doshi, and C. Bear. The Vertica Analytic Database: C-store 7 Years Later. *Proceedings of the VLDB Endowment*, 5(12):1790–1801, 2012.

[37] H. Li, A. Ghodsi, M. Zaharia, S. Shenker, and I. Stoica. Tachyon: Reliable, Memory Speed Storage for Cluster Computing Frameworks. In *ACM Symposium on Cloud Computing (SoCC)*, 2014.

[38] H. Lim, B. Fan, D. G. Andersen, and M. Kaminsky. SILT: A Memory-Efficient, High-Performance Key-Value Store. In *ACM Symposium on Operating Systems Principles (SOSP)*, 2011.

[39] H. Lim, D. Han, D. G. Andersen, and M. Kaminsky. MICA: A Holistic Approach to Fast Inmemory Key-value Storage. In *USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2014.

[40] U. Manber and G. Myers. Suffix Arrays: A New Method for On-line String Searches. In *ACM-SIAM Symposium on Discrete Algorithms (SODA)*, 1993.

[41] Y. Mao, E. Kohler, and R. T. Morris. Cache Craftiness for Fast Multicore Key-value Storage. In *ACM European Conference on Computer Systems (EuroSys)*, 2012.

[42] M. M. Michael. High Performance Dynamic Lock-free Hash Tables and List-based Sets. In *ACM Symposium on Parallel Algorithms and Architectures (SPAA)*, 2002.

[43] J. Ousterhout, P. Agrawal, D. Erickson, C. Kozyrakis, J. Leverich, D. Mazières, S. Mitra, A. Narayanan, G. Parulkar, M. Rosenblum, et al. The Case for RAMClouds: Scalable High-performance Storage Entirely in DRAM. *ACM SIGOPS Operating Systems Review*, 43(4):92–105, 2010.

[44] K. Sadakane. Compressed Text Databases with Efficient Query Algorithms Based on the Compressed Suffix Array. In *International Conference on Algorithms and Computation (ISAAC)*. 2000.

[45] K. Sadakane. Succinct Representations of Lcp Information and Improvements in the Compressed Suffix Arrays. In *ACM-SIAM Symposium on Discrete Algorithms (SODA)*, 2002.

[46] K. Sadakane. New Text Indexing Functionalities of the Compressed Suffix Arrays. *Journal of Algorithms*, 48(2):294–313, 2003.

[47] K. Sadakane. Compressed Suffix Trees with Full Functionality. *Theory of Computing Systems*, 41(4):589–607, 2007.

[48] S. Sivasubramanian. Amazon dynamoDB: A Seamlessly Scalable Non-relational Database Service. In *ACM International Conference on Management of Data (SIGMOD)*, 2012.

[49] M. Stonebraker, D. J. Abadi, A. Batkin, X. Chen, M. Cherniack, M. Ferreira, E. Lau, A. Lin, S. R. Madden, E. J. O'Neil, P. E. O'Neil, A. Rasin, N. Tran, and S. B. Zdonik. C-Store: A Column-Oriented DBMS. In *International Conference on Very Large Data Bases (VLDB)*, 2005.

[50] E. Ukkonen. On-Line Construction of Suffix Trees. *Algorithmica*, 14:249–260, 1995.

[51] M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker, and I. Stoica. Spark: Cluster Computing with Working Sets. In *USENIX Workshop on Hot Topics in Cloud Computing (HotCloud)*, 2010.

Wormhole: Reliable Pub-Sub to support Geo-replicated Internet Services

Yogeshwer Sharma[†], Philippe Ajoux, Petchean Ang, David Callies, Abhishek Choudhary,
Laurent Demailly, Thomas Fersch, Liat Atsmon Guz, Andrzej Kotulski,
Sachin Kulkarni, Sanjeev Kumar, Harry Li, Jun Li, Evgeniy Makeev,
Kowshik Prakasam, Robbert van Renesse[*], Sabyasachi Roy, Pratyush Seth,
Yee Jiun Song, Kaushik Veeraraghavan, Benjamin Wester, Peter Xie.
Facebook Inc., [*]Cornell University
[†]Contact: yogi@fb.com, sharma@cornell.edu

(Operational Systems Track)

Abstract

Wormhole is a publish-subscribe (pub-sub) system developed for use within Facebook's geographically replicated datacenters. It is used to reliably replicate changes among several Facebook services including TAO, Graph Search and Memcache. This paper describes the design and implementation of Wormhole as well as the operational challenges of scaling the system to support the multiple data storage systems deployed at Facebook. Our production deployment of Wormhole transfers over 35 GBytes/sec in steady state (50 millions messages/sec or 5 trillion messages/day) across all deployments with bursts up to 200 GBytes/sec during failure recovery. We demonstrate that Wormhole publishes updates with low latency to subscribers that can fail or consume updates at varying rates, without compromising efficiency.

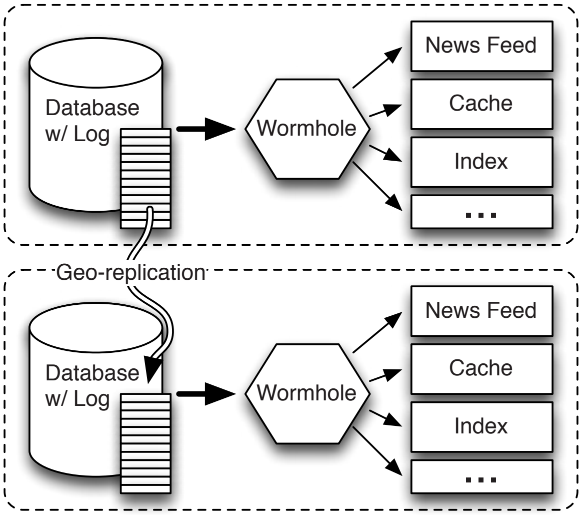


Figure 1: Wormhole reads updates from the data storage system transaction logs and transmits them to interested applications that include News Feed, index servers, Graph Search and many others.

1 Introduction

Facebook is a social networking service that connects people across the globe and enables them to share information with each other. When a user posts content to Facebook, it is written to a database. There are a number of applications that are interested in learning of a write immediately after the write is committed. For instance, News Feed is interested in the write so it can serve new stories to the user's friends. Similarly, users receiving a notification might wish to immediately view the content. A number of internal services, such as our asynchronous cache invalidation pipeline, index server pipelines, etc. are also interested in the write.

Directing each application to poll the database for newly written data is untenable as applications have to decide between either long poll intervals which lead to stale data or frequent polling which interferes with the production workload of the storage system.

Publish-subscribe (pub-sub) systems that identify updates and transmit notifications to interested applications offer a more scalable solution. Pub-sub systems are well studied (see Section 6) with many commercial and open source solutions. However, most existing pub-sub systems require a custom data store that is interposed on writes to generate the notifications for interested applications. This is impractical for Facebook which stores user data on a fleet of sharded storage systems including MySQL databases [18], HDFS [27] and RocksDB [30] across multiple datacenters. Interposing on writes to these storage systems would require modifications across the software stack, which is error-prone and might degrade latency and availability. Writing the updates to a custom data store would also introduce an additional intermediary storage system that might fail.

To address our requirements, we built and deployed

Wormhole, a pub-sub system that identifies new writes and publishes updates to all interested applications (see Figure 1). Wormhole *publishers* directly read the transaction logs maintained by the data storage systems to learn of new writes committed by *producers*. Upon identifying a new write, Wormhole encapsulates the data and any corresponding metadata in a custom *Wormhole update* that is delivered to subscribers. Wormhole updates always encode data as a set of key-value pairs irrespective of the underlying storage system, so interested applications do not have to worry about where the underlying data lives.

Data storage systems are typically geo-replicated with a single master, multiple slaves topology. Wormhole delivers updates to geo-replicated subscribers by piggybacking on publishers running on slave replicas that are close to subscribers. On a write, data is written to the transaction log of the master replica and then replicated asynchronously to the slaves. Wormhole publishers running on the slave can simply read new updates off the local transaction log and provide updates to local subscribers.

Wormhole can survive both publisher and subscriber failures, without losing updates. On the publisher side, Wormhole provides *multiple-copy reliable delivery* where it allows applications to configure a primary source and many secondary sources they can receive updates from. If the primary publisher (which generally is the publisher in the local region) fails, Wormhole can seamlessly start sending updates from one of the secondary publishers. On the subscriber side, an application that has registered for updates can also fail. Wormhole publishers periodically store for each registered application the position in the transaction logs of the most recent update it has received and acknowledged. On an application failure, Wormhole finds where to start sending updates from based on its bookkeeping, maps that to the correct update in the datastore's transaction log and resumes delivering updates.

Wormhole has been in production at Facebook for over three years, delivering over 35 GBytes/sec continuously (over 50 million messages/sec) across all deployments.

Our contributions are as follows:

- We describe the first large scale pub-sub system that delivers trillions of updates per day to subscribers.
- We present a pub-sub system that can run atop existing datastores and provide updates to subscribers.
- We implement *multiple-copy reliable delivery* in Wormhole that allows it to send updates to applications even in the presence of publisher and subscriber failures.
- We allow the datastores to trade-off latency of delivering updates with I/O bandwidth by selecting how much of the disk bandwidth is available for use by Wormhole.

2 Problem

Facebook stores a large amount of user generated data, such as status updates, comments, likes, shares, etc. This data is written to a number of different storage systems depending on several factors such as whether the workload is write optimized or read optimized, what is the capacity versus cost trade-off etc. Moreover, to scale with Facebook's vast user base, these storage systems are sharded and geo-replicated in various data centers.

There are numerous systems that need the newly updated data to function correctly. For instance, Facebook aggressively employs caching systems such as Memcache [19] and TAO [7] so the underlying storage systems are not inundated with read queries. Similarly, Graph Search [12] maintains a index over all user generated data so it can quickly retrieve queried data. On a write, cached and indexed copies of the data need to either be invalidated or updated.

Directing applications to poll the database for newly written data is unscalable. Additionally, writes might be written to any of the different storage systems and applications might be interested in all new updates. Thus, there are a number of challenges that an update dissemination system deployed at Facebook needs to handle:

1. **Different consumption speeds:** Applications consume updates at different speeds. A slow application that synchronously processes updates should not hold up data delivery to a fast one.
2. **At least once delivery:** All updates are delivered at least once. This ensures that applications can trust that they have received all updates that they are interested in.
3. **In-order delivery of new updates:** When an update is received, the application should be confident that all updates prior to the received one have also been received earlier.
4. **Fault tolerance:** The system must be resilient to frequent hardware and software failure both on the datastore as well as the application end.

Challenges 1 and 4 imposed by heterogeneous nature of Facebook's infrastructure, while the others are design choices made based on the nature of applications supported.

3 Wormhole Architecture

In this section, we describe the high level design of Wormhole. Figure 2 shows its main components. In the

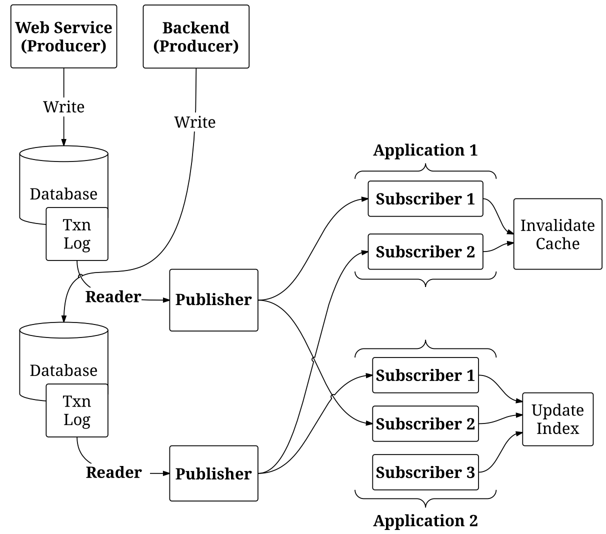


Figure 2: Components of Wormhole. Producers produce data and write to datastores. Publishers read the transaction logs of datastores, construct updates from them, and send them to subscribers of various applications, which in turn do application specific work, e.g., invalidate caches or update indices.

following subsections, we detail specific design choices, their implications, and their role in satisfying the requirements from Section 2.

Data Model and System Architecture. A ***dataset*** is a collection of related data, for example, user generated data in Facebook. It is partitioned into a number of ***shards*** for better scaling, and each update to the dataset is assigned a unique shard. A ***datastore*** stores data for a collection of ***shards*** of the dataset. Each datastore runs a ***publisher***, which reads updates from the datastore transaction log, filters the updates, and sends them to a set of ***subscribers***. Publishers are typically co-located on the database machines so they have fast local access to the transaction log.

In Wormhole publisher, we have support for reading from MySQL, HDFS, and RocksDB, and it is easy to add support for new log types. The ***producers*** of updates write a set of key-value pairs in the log entries in a serialized format that is different for each log type. The publisher takes care of translating the underlying log entries into objects with a standard key-value format, which we refer to as a ***Wormhole update***. One of the keys in the Wormhole updates (called *#.S*) corresponds to the shard of the update: it is written by the producer based on what shard the update belongs to.

Wormhole's subscribing applications are also sharded. An application links itself with the ***subscriber library*** and arranges for multiple instances of itself to be run, called the ***subscribers***. The publishers finds all interested applications and corresponding subscribers via a ZooKeeper-based configuration system. It then divides all the shards to be sent to the application among all of the application's subscribers, eventually calling *onShardNotice()* for notifying subscribers about shards they will be responsible for (see Table 1). It is possible (and likely) that shards belonging to the same publisher might be processed by different subscribers, and shards belonging to different publishers might be processed by same subscriber.

All updates of a single shard are always sent to one subscriber, i.e., they are not split among multiple subscribers. Wormhole arranges for in-order delivery of the updates belonging to any fixed shard: if a subscriber receives an update (say u_1) for a shard (say s_1), then all updates for shard s_1 contained in the transaction logs prior to u_1 must have already been received by the subscriber. Subscribers receive the stream of updates for every shard, which we call a ***flow***. Publishers periodically track ***datamarkers*** per flow after the subscribers acknowledge that they have processed the updates up to new datamarker. A datamarker for a flow is essentially a pointer in the datastore log that indicates the position of the last received and acknowledged update of the flow by the subscriber. Subscribers are assumed to be stateless. In particular, they don't need to keep track of the state of the flow.

Updates Delivery. To get started, a publisher finds applications that want to subscribe to it using configuration files. It constructs flows for these applications corresponding to shards it has in its datastore, constructing one flow for each (application, shard) pair. The configuration can be changed dynamically, for example adding a new application or deleting an old application. This may result in addition or deletion of flows. When a new application is added, it is typically specified which point in the past it wants to start getting updates from. In such case, publishers ensures that it sends updates starting from asked for position.

In steady state, all flows get updates from the same position in the datastore, i.e., the position corresponding to the current time. Hence, Wormhole uses one reader to get the updates, and sends them to all interested flows. In case of error conditions, the publisher needs to restart sending updates from a stored datamarker. For this, the publisher may need to read older updates from the datastore's log. If many flows are recovering simultaneously, a naive implementation of the publisher would read from many positions in the log simultaneously, causing high I/O load. Wormhole clusters flows and creates a reader for each cluster instead, which results in significant I/O savings. Each such reader combined with associated flows is called a ***caravan*** (see Section 3.1). The singleton steady state caravan (or the farthest caravan in case of multiple caravans) is called the ***lead caravan***.

Wormhole needs to load balance flows among the subscribers of an application. We use two modes of load balancing. In the first mode, the publishers use a weighted random selection to choose which subscriber to associate a flow with, so that lightly loaded subscribers are more likely to get more flows. In the second mode, the subscribers use ZooKeeper [13], a distributed synchronization service, to balance load among themselves: If some subscribers get heavily loaded, they redirect some flows to lightly loaded subscribers. Subscribers can also implement a custom load-balancing strategy using these primitives.

After a flow is associated with a caravan and a subscriber, updates are sent over a TCP connection. Publishers save on connection overhead by multiplexing all flows associated with the same subscriber. The datamarkers are moved forward for flows periodically after corresponding subscribers confirm receipt of new updates.

Wormhole provides libraries to make it easy to build new applications. The subscriber library takes care of communication with the publisher, responding to datamarkers, and other protocol details. The subscribers of the application implement an API that is specified in Table 1.

Before moving on to an in-depth description of Wormhole components, we make a few observations. (1) Wormhole is highly configurable, and therefore can be used for diverse applications such as cache invalidations, index updates, replication, data loading to Hive etc. It has pluggable datastore support, allows configurations to be changed on the fly, and has a configurable flow-clustering algorithm (see Section 3.1). (2) Wormhole supports applications that are written in various languages. Currently applications written in C++ and Java are supported. (3) Wormhole is highly reliable and can handle failures of various components: publishers, datastores, subscribers, and even partial network failures.

3.1 Caravan Allocation

As described earlier, Wormhole publisher has the singleton lead caravan in steady state which reads updates from the datastore and sends them to appropriate subscribers. When some subscribers become slow in processing updates, Wormhole creates additional caravans to follow the lead caravan. These additional caravans send past updates to slow subscribers. In addition, flows are dynamically assigned to a varying number of caravans as their datamarkers change in order to optimize for latency and I/O-load. The trade-off between I/O-load and latency can be intuitively seen as follows: If we allow a large number of caravans, we incur higher I/O-load, but we can do a better job of clustering flows whose datamarkers are close, which prevents other flows from having to wait. If we allow very few caravans, flows with very different datamarkers get assigned to the same caravan, making flows which are farther ahead wait for flows that are very far behind. The allocation of flows to caravans is called ***caravan allocation***. Note that the datamarkers of all flows assigned to a caravan must be at least as large as the position of the caravan in the transaction log.

Caravans are periodically split and merged based on the datamarkers of the flows. A caravan is split if the flows on it can be tightly clustered into more than one cluster. Two caravans are merged if they are "close" to each other and reading updates that are nearby in the datastore transaction log. Usually, the non-lead caravans are expected to eventually catch up with the lead caravan and are thus forced to read updates at a rate that is faster than the rate of the lead caravan (typically 1.25 to 2 times faster). In order to prevent overloading the datastore, Wormhole has configuration parameters for the maximum number of caravans, the maximum rate at which a caravan is allowed to read updates, and a maximum cumulative rate at which the collection of caravans is allowed to read updates.

We also dynamically move flows between existing caravans. If a caravan has a flow which is not able to keep up with the speed of the caravan (because the corresponding subscriber is overloaded, for instance) or whose datamarker is far ahead (and can better served by another caravan), we can move the flow. These actions are taken periodically.

3.2 Filtering Mechanism

Wormhole implements *publisher-side filtering*: the application informs publishers of what filters it needs; the publisher only delivers updates that pass the supplied filters. While evaluation of filters places some additional processing overhead on a publisher, it helps conserve both memory and network bandwidth. The efficiency resulting from publisher-side filtering is more pronounced when there are many applications that need only a subset of data.

Filtering is based on the Wormhole update format, which is a set of key-value pairs. Filters are specified as follows: the top-level filter is an "OR" which is a disjunction of finitely many mid-level "AND," each of which in turn is a conjunction of finitely many "basic filters." A basic filter on an update is one of four kinds: (1) Does a key exist, (2) Is the value of a key equal to a specified value, (3) Is the value of a key contained in a specified set (a numeric interval or a regular expression or a list of

Callback for application	**When is the callback invoked by subscriber library**
onShardNotice(shard)	when updates for a new shard are discovered by the publisher, it notifies the subscriber of existence of a new shard
onUpdate(wormholeUpdate)	when a new update is received from the publisher
onToken(datamarker)	when publisher requests acknowledgement that the subscriber has received data up to the new datamarker
onDataLoss(fromMarker, toMarker)	when the publisher realizes that some data for the flow was not sent

Table 1: API that Wormhole subscribers implement to get updates from publishers. The callbacks specified are run when an event of a specific type happens. *onShardNotice()* is called when a subscriber is notified of a new flow corresponding to a new shard. Once the flow is established, *onUpdate()* is called for each received update. When the publisher sends a new datamarker asking for acknowledgement of received data, *onToken()* is called. When the subscriber has processed all updates up to the supplied datamarker, it is passed back to the publisher as an acknowledgement. In the rare event that truncation of logs by the underlying datastore results in data loss for an application because the application was further behind than the truncated log position, *onDataLoss()* callback is called to notify the subscriber of data loss event.

elements), or (4) negation of any of the previous three.

This generic filtering system is flexible enough for Facebook's various applications. For example, for the cache invalidation applications we use the filter *[topic = aq] OR [mcd key exists]* (*mcd* specifies keys to invalidate in Memcache). For index update services, we use more complex filters such as *[tableName in (t1, t2)] OR [(associationType = a1) AND (shard in 1-5000)].*

3.3 Reliable Delivery

Reliability is an important requirement for Wormhole. For example, one missed update could lead to permanent corruption in a cache or index of a dataset. Wormhole supports two types of datasets: *single-copy datasets* and *multiple-copy datasets*. The latter indicates a geo-replicated dataset. Accordingly, Wormhole supports both ***single-copy reliable delivery*** (***SCRD***) and ***multiple-copy reliable delivery*** (***MCRD***). For SCRD, Wormhole guarantees that when an application is subscribed to the single copy of a dataset, its subscribers receive *at least once* all updates contained in that single copy of the dataset. The updates for any shard are delivered to the application *in order* that they were stored in the transaction logs: delivery of an update means all prior updates for that shard have already been delivered. For MCRD, applications are allowed to subscribe to multiple copies of a dataset at once, and when they do so, Wormhole guarantees that its subscribers receive *at least once* all updates contained in any subscribed copy of the dataset. The updates for any shard are, again, delivered *in order*. There is no ordering guarantee between updates that belong to different shards. Ordering guarantee for updates within shards suffices for most purposes, since updates corresponding to one entity (e.g., a Facebook page, or a Facebook user) reside on the same shard.

Note that these guarantees do not hold if an update is not available in the datastore log at the time application is ready to receive updates (because datastore might have truncated its logs). Typically datastore logs retain updates for 1–2 days, and an application that falls behind by more than that may thus miss updates (and notified by *onDataLoss()* callback, see Table 1). In our experience, when applications do fall behind because of machines or network failures, monitoring alarms become active and remediation is done quickly. Hence, it is rare for the applications to fall behind by more than a few hours.

In rest of this section, we first show how Wormhole uses datamarkers to provide SCRD, and then how it is extended to MCRD.

3.3.1 Single-Copy Reliable Delivery (SCRD)

Wormhole leverages the reliability of TCP: while a subscriber is responsive, TCP ensures reliable delivery. Wormhole does not use application layer acknowledgements for individual updates—we found it resulted in heavy bandwidth usage and lowered throughput. Instead, for every flow, a publisher periodically sends a datamarker (current position in the datastore log) interspersed with updates. The subscriber acknowledges a datamarker once it has processed all updates before the datamarker. The acknowledged datamarkers are stored on the publisher side in persistent storage. Since the publisher can send updates to a flow only if both the datastore and the datamarker are available, it makes sense to store them together.

These stored datamarkers help the publisher achieve SCRD. When a subscriber becomes available after subscriber or network failure, a publisher uses previously acknowledged datamarkers as starting points for sending updates, hence not missing any update.

The decision of sending datamarkers only periodically results in higher throughput in the normal case. But in the case of a recovery, when publisher starts sending updates from previously acknowledged datamarker, some updates may be received more than once. This is not a problem in most cases. Multiple deliveries of cache invalidation updates do not violate correctness, and other applications can easily built logic to remove duplicates. Wormhole also provides an interface to allow subscribers to send datamarkers to publishers in order to reduce the probability of duplicate delivery.

The average size of datamarkers is less than 100 bytes and they are sent only once every 30 seconds. Hence network overhead from datamarkers is small: 0.0006%–0.0013% of the traffic. This overhead can be reduced by increasing the period between datamarkers, but doing so results in a higher overhead when recovering. With 30 seconds period, 0.2 seconds of updates are resent for every 10 minutes of updates on average because of failures and recoveries, resulting in 0.03% network overhead.

We also support a mode where applications can choose to get only real time data (hence not requiring caravans for old updates), or data that is at most a certain time old. While it does not guarantee reliable delivery, this mode is frequently used for developing and testing applications.

3.3.2 Muliple-Copy Reliable Delivery (MCRD)

Most datasets at Facebook are replicated, allowing for higher availability of updates. In MCRD, when a publisher doing single-source reliable delivery to an application fails permanently, e.g., because of hardware failure, we would like a publisher running on replicated datastore to take over and do single-source reliable delivery to the application starting from where the failed datastore stopped sending updates. In essence, MCRD is "SCRD with publisher failover." There are, however, several challenges in extending the SCRD guarantee to MCRD:

(1) The datamarkers for flows are stored in persistent storage by the publisher, which is typically colocated with the datastore host. When the host fails, we lose the datamarker even though the updates might be available elsewhere.

(2) The datamarker for a flow is a pointer into the logs of the datastore. It is usually a filename of the log and byte offset within that file. Unfortunately, datamarkers represented this way are specific to the particular replica and it is not straightforward to find the corresponding position in a different replica. For example, in MySQL, binary log names and offsets are completely different for different replicas.

(3) For simplicity, publishers are independent entities in MCRD case, and they do not communicate to each other. For ease of operations, we would still like a solution that minimized the communication between publishers.

We address these challenges in the following ways. First, MCRD publishers store datamarkers in ZooKeeper, a highly available distributed service.

To overcome the problem of replica-specific datamarkers, we introduce ***logical positions***—a datastore agnostic way to identify updates such that copies of the same update in different replicas have the same logical position. A logical position uniquely identifies an update in a dataset using a (monotonically non-decreasing) ***sequence number*** and an ***index***. The updates are assigned logical positions by the publisher. When the sequence number of consecutive updates are equal, which can happen if the datastore does not natively support sequence numbers and we use timestamps of updates as sequence numbers, they are assigned monotonically increasing indices starting at 1 so they have unique logical positions. Since caravans still need datastore positions to start reading logs, a data structure called ***logical positions mapper***, or simply ***mapper***, maps logical positions from datamarkers to datastore-specific positions.

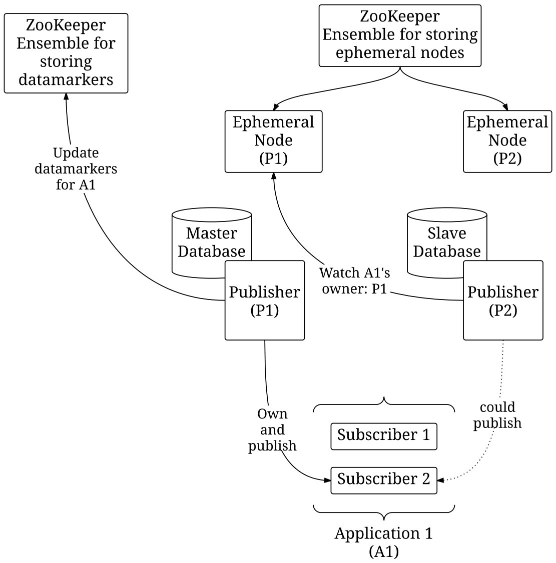


Figure 3: Architecture for the failover of publishers in MCRD. Multiple publishers (in this case P1 and P2) have the same data to publish to application (A1), but only one of them (P1) owns and publishes. Each publisher has an ephemeral node corresponding to it, which non-owner publisher watches in case the owner fails (P2 watching P1). The owner publisher updates datamarkers in ZooKeeper. If P1 fails, P2 will notice the disappearance of P1's ephemeral node and will start owning the flows for the application.

Finally, to reduce the communication among publishers, we note that MCRD publishers need to be aware only of their peers serving the *same* updates. Therefore, a publisher needs to only communicate to as many publishers as there are replicas. At any time, one peer *owns* a flow and other peers *watch* the owner for any changes. The owner publisher is responsible for sending updates and recording logical datamarkers in ZooKeeper. All other peers keep track of the owner (or ZooKeeper's *ephemeral nodes* [13] corresponding to the owner) and take over the ownership if the owner fails. This architecture is illustrated in Figure 3.

4 Workload and Evaluation

Wormhole has been in production at Facebook for over three years. Over this period, Wormhole has grown tremendously from its initial deployment where it published database updates for cache invalidation system. Today, Wormhole is used by tens of datasets stored on MySQL, HDFS and RocksDB. Across these deployments, Wormhole publishers transport over 35 GBytes/sec of updates at steady state across 5 trillion messages per day. Note that this update rate isn't the peak for the system—in practice, we have found that Wormhole has transported over 200 GBytes/sec of updates when some applications fail and need to replay past updates. We have found that designing Wormhole to run alongside sharded datastores has allowed us to scale the system horizontally by bringing up new publishers on the datastore machines.

To keep Wormhole from hurting datastore performance, a typical constraint in production is to not start too many readers to read updates from datastores. As a result, the historic average of number of caravans used by Wormhole publishers in production is just over 1 (≈ 1.063).

Our evaluation of Wormhole focuses on our production deployment and a few synthetic benchmarks that illustrate Wormhole's characteristics. We focus on the following metrics.

Scalability and Throughput: What is Wormhole's publisher and subscriber throughput? How well does it scale with the number of applications subscribing to updates?

Efficiency: Do caravans reduce load on datastores? How well does Wormhole trade-off I/O efficiency with latency in delivering updates?

Latency: What is the typical latency for delivering updates?

Fault Tolerance: How well does Wormhole handle the failure of publishers, subscribers and even a whole datacenter?

4.1 Scalability

4.1.1 Scaling with the number of applications

This experiment evaluates how a single publisher scales with an increasing number of applications.

Methodology. We start one publisher configured to use 4GB of memory and 32 CPUs clocked at 2.6 GHz. The datastore is filled with 5 GBytes of past updates from production traffic with updates having a mean size of 1 KBytes. We run 20 experiments, parameterized by number of applications $n = 1, 2, \ldots, 20$. For each n, we configure n applications to receive updates from the publisher. Each of the n applications have one subscriber, which simply receives all 5 GBytes of updates and increments a counter indicating the number of updates received. The publisher is configured to use only one replay caravan. (The lead caravan is at the end of the logs and not relevant for this experiment.) We measure how long it takes the publisher to send all updates to all n applications (i.e., time to replay), and the rate of sending updates (i.e., throughput).

Results. Figure 4(a) plots the time taken to deliver all updates to all applications. We see the time taken to deliver all queued updates grows linearly with the number of applications. This linear growth stems from the publisher having to schedule each update for delivery to a subscriber of each application.

Figure 4(b) plots the average throughput of the publisher over the time of delivery of all updates. We find that the throughput increases with increasing number of applications before it levels off at just over 350 MBytes/sec. This bottleneck is caused by a lack of parallelism in our publisher, which has not been optimized because Wormhole publishers are typically co-located with production databases and are not allowed to use many cores.

Note that the goal of this experiment is to stress test the publisher by configuring each application to subscribe to all updates. This is in contrast to our production setup where each application typically gets a filtered subset of updates. In our production set up, it is common for publishers to deliver updates to many tens of applications in steady state.

4.1.2 Subscriber throughput

We now turn our attention to the throughput of the subscriber. In this experiment, we stress test a single subscriber by increasing the number of updates the subscriber is configured to get. This is done by increasing the number of publishers whose updates the subscriber is configured to get.

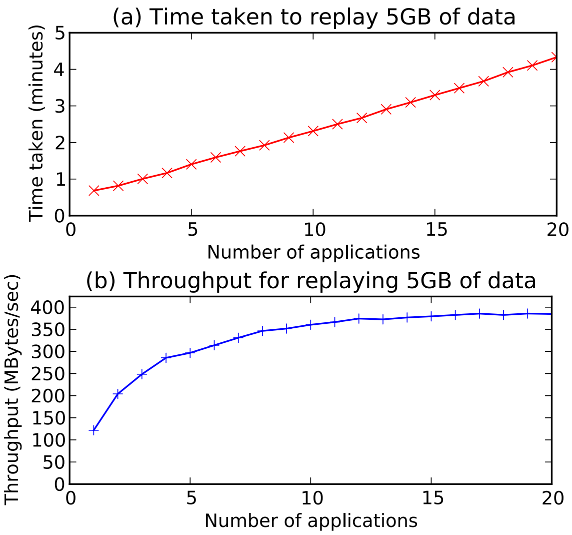


Figure 4: Wormhole publisher delivering updates to varying number of applications from 1 to 20. Panel (a) shows time taken to deliver all updates using a single replay caravan. Panel (b) shows the average throughput of the publisher during delivery of all updates.

Methodology. We start one application using one subscriber configured with 4 GBytes of memory and 4 Intel 2.80 GHz CPUs. We configure this subscriber to get a fraction of updates from a production dataset that averages 500 bytes per update. We periodically increase the fraction of updates the subscriber is configured to get by increasing the number of shards from the dataset that the application is interested in. This is done approximately every 15 minutes. We measure the running average of throughput and latency of the subscriber during the experiment.

The measured data is plotted in Figure 5. Both (a) and (b) show the number of shards whose updates the subscriber is getting over the 150-minute experiment. Note that the step increments in this graph at minutes $\approx 80, 95$, and 110 is by manually changing the configuration of the application to be subscribed to more shards from publishers. Figure 5(a) shows the average throughput of the subscriber for each one minute interval. Figure 5(b) shows the average latency of updates delivery for each one minute interval. Note that this latency is end-to-end—the difference between the time at which update was delivered to the subscriber and time at which it was written to the datastore where publisher is reading. Also note that during the time of low latency, the sum total of send-throughputs of all publishers to the subscriber is equal to the receive-throughput of the subscriber.

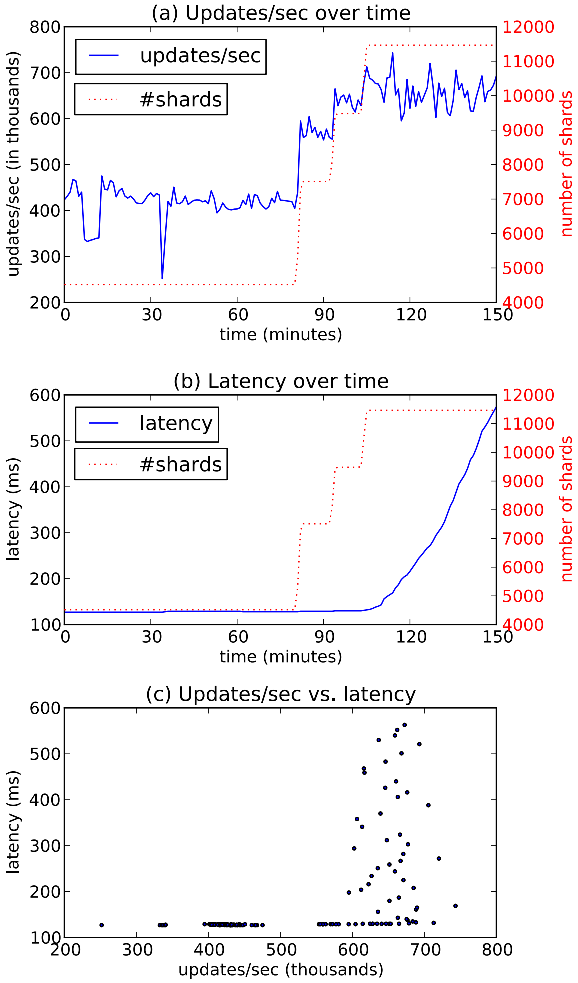


Figure 5: Running average of throughput and latency of subscriber that gets increasingly larger amount of updates from publishers. Increasing amount of updates is shown in (a) and (b) by step increase in number of shards whose updates the subscriber gets. This increase is done manually at minutes $\approx 80, 95$, and 110. Panel (a) shows the throughput in updates/sec, which jumps with jump in number of shards. Panel (b) shows the average latency of delivery of updates, which remains constant up to the throughput limit of the subscriber. Panel (c) combines (a) and (b) by plotting throughput versus latency of each one minute interval shown in (a) and (b). It shows that the latency of updates delivery remains low up to a limit, and increases in an unbounded manner after that.

Note that the throughput jumps when number of shards jump, which is expected because more publishers start sending updates to the subscriber. Despite this increased throughput, the average latency of updates remains constant at 150ms. But the final jump in throughput around minute 110 is not sustained—the throughput hovers around 600,000 updates/sec. Also, the latency starts increasing without limits at minute 110, since the subscriber is not able to handle all the updates. This leads to publisher having to allocate a different caravan for sending updates again. This time is counted against latency of updates, which results in higher latency for updates that are resent. This lack of ability of subscriber to process more than 600k updates/sec is easily seen in Figure 5(c), which is a convenient combination of the previous two: For each one minute interval, it plots the average throughput versus average latency during that one minute interval. After the throughput hits 600k updates/sec, the latency keeps increasing without any further increase in the throughput.

Results. Wormhole subscriber can sustain a high throughput of over 600,000 updates/second, without imposing a latency penalty. If we try to push more updates than that, the subscriber starts dropping updates, which results in publishers having to resend them and causing large latency.

4.1.3 Publisher throughput in production

A previous experiment showed the throughput of the publisher for replaying updates for data that was not serving production traffic (Figure 4). We now evaluate the typical speed of recovery for application in Facebook's production environment and show that Wormhole is capable to serving high throughput in bursts when applications fall behind.

We consider a production deployment of Wormhole with publishers delivering updates to a cache invalidation application. For this application, any delay in delivering updates results in stale cache data.

Methodology. We evaluate 50 publishers over a 24-hour period and report what is the average throughput of the publishers, and what is the maximum throughput of the publishers observed during the experiment. To count towards the maximum, the throughput had to be sustained for at least one minute. We plot the average throughput versus maximum throughput in Figure 6.

Results. The main take-away from this experiment is that in Facebook's production environment, Wormhole publishers are capable of sustaining throughput that is more than 10 times their average throughput. This result is important since it is common for applications to fall behind. When requested, the publisher must be able to help the application recover quickly by sustaining high throughput for short bursts of time.

Note that in the above production environment, the highest throughout a caravan can achieve was artificially capped in the configuration of these publishers so that Wormhole does not adversely affect the performance of the underlying datastore. In the absence of such constraints, Wormhole is capable of higher throughput but may result in worse datastore performance for other clients.

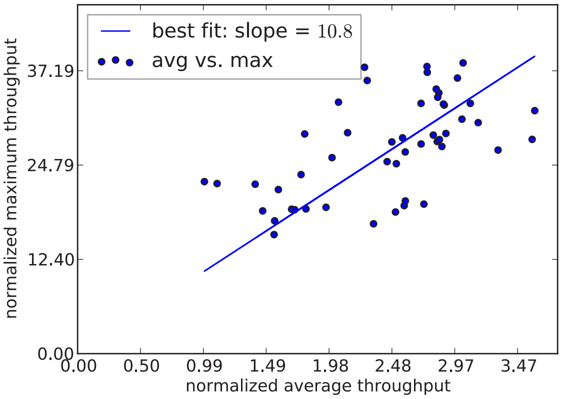


Figure 6: Sustained maximum throughput of publisher versus average throughput over a period of 24 hours for a set of 50 publishers. The throughputs are normalized by making minimum average throughput equal to 1 unit.

4.2 Efficiency

In this section, we evaluate the efficiency achieved by Wormhole by using caravans.

4.2.1 Trading off latency for I/O during recovery

After a failure, multiple applications might fall behind and request updates from different points in time in the past. Wormhole has a choice of creating many caravans starting at different positions in the datastore logs, or using fewer caravans and clustering flows from applications. On one extreme, if multiple caravans are spawned, applications can receive updates immediately, resulting in low latency of update delivery but high read amplification (i.e., updates being read multiple times by different caravans). On the other extreme, if we are allowed to start only one caravan, applications that are further along have to wait for lagging applications to recover before getting updates. This results in higher latency for applications that are up to date. We simulate this scenario in evaluating Wormhole's trade-off between I/O and latency.

Methodology. We start a single publisher on a datastore that has 20 GBytes of updates. To simulate multiple applications that fall behind by different amounts in production, we subscribe this publisher to 10 applications whose datamarkers are equally distributed across the 20 GBytes of updates. Therefore, each application wants to get a progressively smaller tail-end of the updates in the datastore: first application wants to get the whole 20 GBytes, the second application wants to get the last 18 GBytes, and so on, and tenth application wants to get the last 2 GBytes of updates.

Updates from the publisher to all applications can be sent using different number of caravans (which results in different read amplification factors). We run the experiment in 7 iterations, with different maximum number of caravan Wormhole publisher is allowed to use: 10, 7, 5, 4, 3, 2, and 1. For each iteration, we measure how much data is read collectively by all caravans and divide it by total amount of data in the datastore to get the read amplification factor of the iteration. This is plotted on the *x*-axis in Figure 7.

We measure the latency as follows. The latency of one update is measured as the difference between time of receipt of update by application and time of commit of the update in the datastore. The latency of one application is the average latency over all updates it received. The latency plotted on *y*-axis is the average latency of all 10 applications. Since we are replaying past data spanning several minutes, the latency as measured above is expected to be in minutes.

Each data point in Figure 7 corresponds to one iteration of the experiment, indicating the read amplification factor and the average latency of all applications. The label for the data point corresponds to the maximum number of caravans allowed in that iteration.

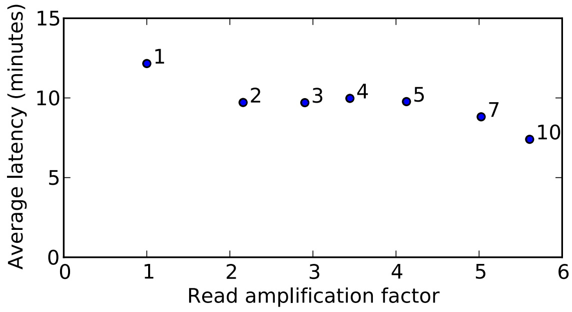


Figure 7: Read amplification factor versus average latency of delivering updates to 10 applications (see text for more explanation). Different data points are for varying number of maximum allowed caravans. The latency is averaged over 10 applications whose datamarkers are scattered evenly across the 20 GBytes of single datastore updates.

Results. The main result of this experiment is to demonstrate that Wormhole can trade off the load on datastore for latency of serving updates. Figure 7 shows that by increasing read amplification on the datastore, Wormhole is able to reduce the average latency of updates by up to 40%.

Note that each addition caravan does not reduce the latency of updates. This is an artifact of how we assign flows to caravans. We believe this can be improved with a different caravan allocation algorithm. This is an active direction for future research, see Section 7.

4.2.2 Updates delivered versus updates read

Methodology. We evaluate how many bytes of updates Wormhole publishers read for each byte of updates sent to all applications. A lower number for this metric indicates Wormhole publisher puts little load on datastores in order to send updates to many applications.

We use measurements from a production deployment that is used to replicate (cache) data across datacenters. There are multiple publishers in the datacenter we are considering, and 6 applications subscribed to the dataset corresponding to these publishers (the number of publishers and subscribers is not relevant to this discussion). The publisher and subscribers are in geographically distributed locations (publisher on the east coast, subscribers on east and west coasts of the US, and Europe). Over a period of 48 hours, we observe the number of bytes collectively sent by these publishers to 6 applications, number of bytes read from datastores by the publishers, and how many caravans were used to read those bytes. These metrics are collected every 1 minute, and the collected value is average of the values since previous collection. The results appear in Figure 8.

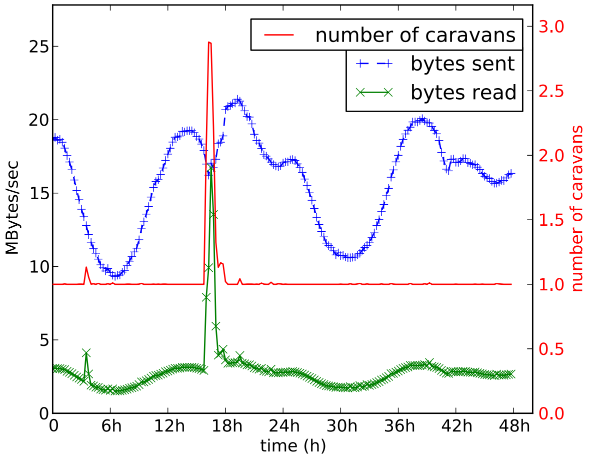


Figure 8: The amount of data read from datastores and sent to applications by Wormhole, and the number of caravans used to do so. The number of caravans, which is averaged over one minute intervals and over all involved publishers, can be fractional.

Results. We see that for a vast majority of time, the ratio of bytes read over bytes sent remains as low at $1/6$. This shows the efficiency of Wormhole in reading updates few times, before sending them to many applications.

There are small spikes in bytes read graph, which corresponds to times when some subscribers of some applications have errors, and hence fall behind. During these spikes, the number of caravans go up accordingly.

Note the large spike in the middle, where the bytes read becomes close to bytes sent (the ratio very close to 1). This happens because many subscribers of one application (out of 6) fall behind because of a systemic error, and all publishers have to read past updates to send past data to the application. Note that even though the number of caravans becomes (only) 3 (for 6 applications), the ratio of bytes read to bytes sent comes close to 1 (instead of 0.5 or $3/6$). The reason is that the two extra non-lead caravans read updates at a higher rate than the lead caravan.

This experiment suggests Wormhole is efficient, even in a large deployment, in reading updates few times and sending them to many applications, and in its ability to recover from applications failures.

4.3 Low Latency

Methodology. We examine the latency experienced by updates in Wormhole. We use the data from one of Facebook's production deployments: we pick one (random) publisher that is sending updates to a cache invalidation application, and observe it over a period of few hours to get a sample of 50,000 updates. The cache consistency application itself runs on hundreds of machines, but the updates from one publisher are distributed to a small number of them (1–3). The average size of updates is 500 bytes. The publisher and subscribers reside in the same datacenter for this setup: the latency between publisher and subscribers (measured via ping time) is low (under 1 ms).

The latency of updates is measured as follows: the publisher writes the (millisecond) timestamp for each update indicating when update was written to the datastore (which could be much earlier than when it was read by the publisher). When the subscriber receives it, it computes the difference of clock time and the timestamp written in the update. The clock skew between the publisher and the subscriber was measured to be less than 1ms. Note that this latency does *not* measure the time taken by the cache consistency application to invalidate the cache.

Results. The cumulative distribution function of latencies of 50,000 updates appear in Figure 9. Over 99.5% of the updates are delivered in under 100ms. Note that there is a long tail of updates that can take as long as 5 seconds. This can happens when an update is sent by a non-lead caravan. In such case, it includes the time that the corresponding flow spends waiting to be assigned to a caravan.

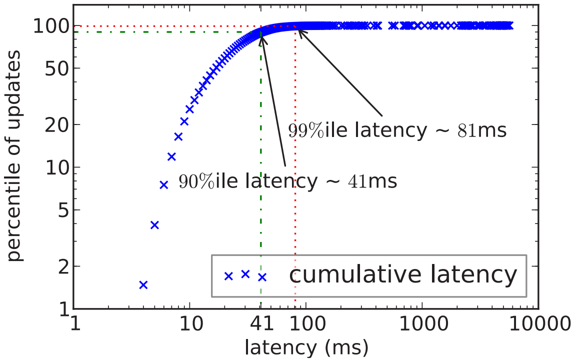


Figure 9: Latency of updates over a period of time, sent from one publisher to local (datacenter) application.

4.4 Reliability and Fault Tolerance

In this section, we evaluate the efficacy of Wormhole in providing multiple-copy (and single-copy) reliable delivery (MCRD and SCRD) by doing a failover for all publishers within a datacenter, by causing single publisher failure, and by causing subscriber failures.

4.4.1 MCRD at large scale

To demonstrate MCRD at large scale, we picked one application that was receiving 300 MBytes/sec of updates from production. We simulated the failure of a datacenter by changing the configuration of the application to get updates from secondary datacenter, instead of the primary datacenter. The ping time between secondary datacenter and application subscribers was 15ms.

Averaged over multiple failovers, it took Wormhole approximately 5 minutes to transfer all traffic from primary to secondary datacenter. A majority of this time is spend during timeouts (for example, it takes one minute before a machine is considered not reachable).

Note that in the case of actual failure of datacenter, the application would not be receiving any data during this time, and receive a burst afterwards when the updates are being sent from the secondary datacenter (qualitatively similar to the bursts in Figure 10(b)).

4.4.2 Reliability under single publisher failure

In this section, we evaluate how Wormhole handles the failures of publishers for SCRD and MCRD.

Methodology. We select two datastores that are replica of a datastore in production. We start one publisher each on them, which serve as peer publisher for MCRD. Call them primary and secondary. We then start two applications: the first one requires multiple-copy reliable delivery of updates, from either primary or secondary, with the preference of primary first. The second application requires single-copy reliable delivery of updates, from the primary publisher only.

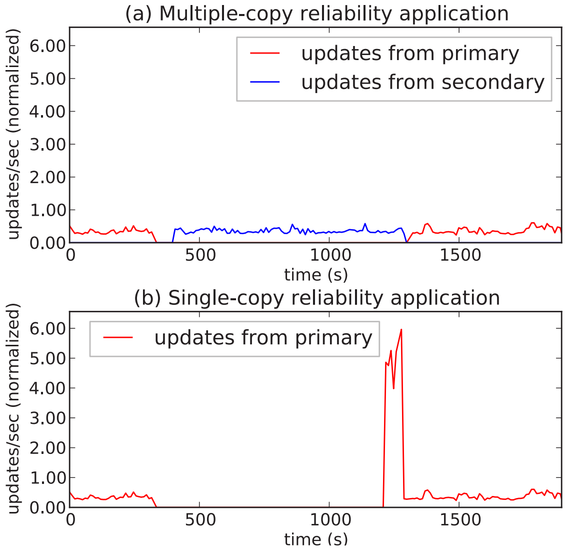


Figure 10: Effect of publisher failure on delivery of updates. Panel (a) shows an MCRD application that is configured to receive updates from either the primary publisher or the secondary publisher. Panel (b) shows an SCRD application that is configured to receive updates only from the primary. The primary publisher is failed close to second 300, and restored at second 1300. In (a), the application starts getting updates from secondary, while in (b), the application has to wait until primary is restored, resulting in higher backlog.

After both applications are receiving data from the primary, we simulate the failure of primary by killing the publisher process on primary (seen by disappearance of red line in Figure 10(a) and (b). We restart the primary publisher 15 minutes after its failure. Note that the secondary publisher runs without any failures.

Figure 10(a) plots the number of updates delivered to the MCRD application from primary and secondary publisher. Figure 10(b) plots the number of updates delivered to the SCRD application from the primary publisher.

Results. This experiment demonstrates the reliable delivery guarantee of Wormhole, showing that both applications survive the failure of primary publisher, albeit in different ways.

The MCRD application starts receiving updates from the secondary publisher within 60 seconds of primary failing. This time comes from the timeout we use to indicate that a publisher is not available any more. When the primary publisher is restored, the MCRD application seamlessly switches to it.

Note the large spike in the updates received for SCRD application. When the only publisher that could deliver updates to it is restored, it sends the backlogs of updates at a higher throughput, and then restores the application to normal state.

4.4.3 Subscriber failures and load balancing

We evaluate Wormhole's capability to balance load among the subscribers of the same application. As described earlier, we use two methods to distribute flows among subscribers: (1) the publisher uses a probability density function to assign flows to subscribers that have relatively fewer flows, and (2) subscribers use a ZooKeeper based load balancing method to provide hint to publisher in choosing subscribers for flows. In the latter method, ZooKeeper based service assigns shards to subscribers, and rebalances periodically when it discovers that the assignment is not balanced. The period is configurable, but typically once per minute.

Methodology. In this experiment, we consider a large number of publishers spanning many shards, say *n*, delivering updates to a production application that also has a large number of subscribers, say *m*. We first use one algorithm to balance load among subscribers and then use the second algorithm.

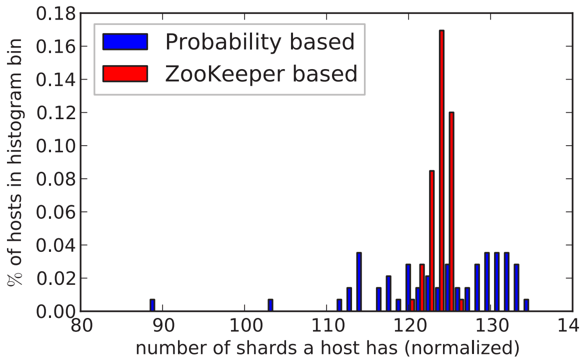


Figure 11: The histograms for distribution of flows among subscribers. The *y*-axis shows the fraction of subscribers that have number of flows that fall in the horizontal *x*-axis bin. The two histograms show the spread for probability based distribution and ZooKeeper based distribution.

To measure the efficacy of the load balancing algorithm, we find out how many shards each subscriber is subscribed to. (The sum of these numbers is always *n*, since there are *n* shards in total.) We normalize this number and put subscribers in 50 bins according to normalized number of shards assigned to them. This is the *x*-axis in Figure 11. We draw the histogram in Figure 11 where *y*-axis shows the fraction of subscribers that have the number of shard subscribed to them on the *x*-axis. A large spread (as in blue histogram) shows that some subscribers have very few shards while others have very many.

Results. As seen in Figure 11, for the ZooKeeper based allocation policy, the spread of number of shards is tightly concentrated. Most subscribers have between 122 and 126 (normalized) shards. On the other hand, a random distribution assigns as few as 88 and as many as 135 shards to subscribers.

5 Operational Challenges

In this section, we discuss the challenges we have faced in running Wormhole at the scale of Facebook's infrastructure, how we addressed them, and how the system has evolved in a way to make it easy to catch and fix problems.

First, the impact of malfunctioning of Wormhole affects some Facebook users much more than others. For example, suppose Wormhole publishers are malfunctioning on 1% of datastore machines so that 1% of the cache is stale. This would cause 100% of cached data for 1% of the users to be stale—not 1% of the cached data for 100% of the users. This makes the reliability of Wormhole publishers all the more important.

Wormhole must run on all production-ready datastore machines, a large set that keeps changing as machines are brought in to and taken out of production. In our experience with a central deployment system we used early on, it is challenging to keep Wormhole publisher running on exactly this set of machines and no other; a central deployment to a large set of machines is likely to result in some mistakes. We decided to switch to a distributed deployment system based on *runsv* [20] while trying to minimize dependencies outside of the local machine. We run a lightweight ***Wormhole monitor*** on each datastore machine. The monitor periodically checks the configuration system (which indicates the machines that are in production) and based on that determines whether to run a publisher or not, and if so, with what configuration. This decentralized system has been significantly more reliable and easier to use.

In order to make debugging the publisher and fixing problems easier, we can change configurations on-the-fly. For example, the maximum rate at which a caravan can send updates can be changed without restarting the publisher. The publisher also implements a *thrift interface* [4]—we use it to access state (e.g., what are the datamarkers of all flows on it) and gather monitoring statistics that the publisher collects and aggregates every 30 seconds (e.g., the rate of updates sent to all applications). The publisher collects over 100 such monitoring statistics that we use to determine the health of the group of publishers. This interface can also be used to give commands to the publisher to override some decisions manually (e.g., reassign all flows to caravans in order to improve I/O utilization), although this is rarely required.

Wormhole's resource utilization depends not only on its own health, but the health of all the subscribing applications. The difference between best-case resource utilization (one caravan in case all applications are current) and worst-case resource utilization (maximum number of caravans) can be large. We have to plan resources for such worst-case operability, e.g., having a resource limit that is higher than normal operating range for Wormhole.

6 Related Work

The pub-sub systems have been an active area of research for many decades. Many pub-sub systems, message buses (topic-based pub-sub systems), and P2P notification systems have been developed [5,6,9,11,16,17,24,25, 28,31] that shares similar goals to Wormhole. Most solutions, though, use *brokers*—intermediate datastores that store and forward updates. These brokers offload the responsibility of forwarding events to subscribers from the datastores. They can provide reliability by buffering updates for slow subscribers, while providing low latency to fast subscribers. However, these solutions are undesirable for us as they require additional infrastructure for brokers: we do not need brokers to buffer updates as our datastores already provide reliable logs in form of transaction logs. Also, brokers can add significant latency to message delivery, particularly if used hierarchically for large scale systems.

Below we consider some of the best known publicly available products.

SIENA [8] is a wide-area content-based pub-sub service. Much of the focus in SIENA is on its specialized content-based routers, while Wormhole uses stock network routers while filtering happens directly at the publishers. SIENA does not support replicated data sources, and has not been demonstrated at a scale or load near Facebook's.

Thialfi [1] is Google's cloud notification service that addresses a similar problem to Wormhole, namely the invalidation of cached objects. Thialfi is geographically distributed and highly reliable, even in the face of long disconnections. However, its clients are applications running in browsers on end-users' mobile phones, laptops, and desktops, not applications within Google itself, such as caching and indexing services. The workloads on Thialfi are, accordingly, very different from Wormhole. In particular, Thialfi is not concerned about I/O efficiency on data sources. Thialfi also sends only version number for the data to subscribers, and coalesces many updates into most recent one. Wormhole, on the other hand, sends all updates, and each update contains more information needed by the application, not just the version number. This is for two reasons: (1) By sending data, we can do cache refills, instead of just cache invalidations, and (2) Wormhole is used for wider purposes, such as RocksDB replication, that need each update being delivered (based on statement replication).

Kafka [15] is LinkedIn's message bus, now open source and maintained by Apache. It has a topic-based pub-sub API. Like Wormhole, it uses ZooKeeper to keep track of how many events particularly subscribers have consumed. As in Wormhole, data sources are sharded. At LinkedIn, it is used to distribute various real-time logging information to various subscribers. Kafka can lose messages in case one of its message brokers suffers a failure. Recent benchmarks puts the speed at which Kafka can transport messages at about 250 MBytes/sec, orders of magnitude below Wormhole's production load, but Kafka's throughput can be improved by sharding differently.

Hedwig [3] is a topic-based Apache pub-sub system with an emphasis on handling many topics and providing strong reliability, not on many publishers or subscribers or on high message load. Many pub-sub systems focus on expressive filters and implement sophisticated ways to filter updates [10, 21, 26].

Messages buses like IronMQ [14] and Amazon SQS [2] are hosted, and cannot be installed in local infrastructure. Beanstalkd [22] and RabbitMQ [23] are popular efficient open-source message buses. Beanstalk supports reliability but is specialized to be used as a collection of task queues. Like Wormhole, RabbitMQ can scale to multiple datacenters and is particularly efficient for small messages. Neither supports replicated data sources, or have been been demonstrated to support the scale of Facebook's workloads.

TIBCO Rendezvous [29] is perhaps the most used and advanced commercial message bus. While it has impressive features and performance, it does not support replicated datastores out-of-the-box. Rendezvous also needs additional storage for messages, which grows with the *reliability interval*, during which message can be retransmitted. The Rendezvous daemon does not guarantee delivery to components that fail and does not recover for periods exceeding the reliability interval, which is a different order of magnitude (typically 60 seconds) than the failure durations of Wormhole components (sometimes many hours).

7 Future Work

As Wormhole continues to support the growing amount of traffic flowing through it, there is need for different features to support the load and diversity of use-cases. Because of the growth in number of applications, we are working to provide differentiated guarantees to applications based on how important fresh data and latency is to them. For example, if an application can afford to get data that is stale up to a few minutes, updates for that application can be batched and compressed to save network bandwidth. We are also working on making it easy to swap in and out various caravan allocation policies in the Wormhole publisher, and measure their efficacy for different workloads.

8 Conclusion

This paper describes Wormhole, a pub-sub system developed at Facebook. Wormhole leverages the transaction log of the storage system to provide a reliable, in-order update stream to interested applications. We have demonstrated that Wormhole scales to support multiple data storage systems and can guarantee delivery in the presence of both publisher and subscriber failure. Our production deployment of Wormhole transfers over 35 GBytes/sec in steady state (over 5 trillion messages per day) from geo-replicated datastores to multiple applications with low latency.

Acknowledgements

We would like to thank Nathan Bronson, Daniel Peek and Jonathan Kaldor for reading earlier drafts of this paper and helping improve it. We would also like to thank the users of Wormhole at Facebook, who provided us with the workloads and insights to help design, implement, and scale Wormhole. We are grateful to anonymous NSDI reviewers and our shepherd Atul Adya for their feedback and detailed comments which helped improve the paper.

References

[1] A. Adya, G. Cooper, D. Myers, and M. Piatek. Thialfi: A client notification service for internet-scale applications. In *Proc. 23rd ACM Symposium on Operating Systems Principles*, pages 129–142, 2011.

[2] Amazon Web Services, Inc. Amazon simple queue service. http://aws.amazon.com/sqs/, 2014.

[3] Apache Software Foundation. HedWig. https://cwiki.apache.org/confluence/display/BOOKKEEPER/HedWig, 2014.

[4] Apache Software Foundation. Thrift. https://thrift.apache.org/, 2014.

[5] R. Baldoni, C. Marchetti, A. Virgillito, and R. Vitenberg. Content-based publish-subscribe over structured overlay networks. In *Distributed Computing Systems, 2005. ICDCS 2005. Proceedings. 25th IEEE International Conference on*, pages 437–446. IEEE, 2005.

[6] R. Baldoni and A. Virgillito. Distributed event routing in publish/subscribe communication systems: a survey (revised version). Technical report, Dipartimento di Informatica e Sistemistica, Università di Roma la Sapienza, 2006.

[7] N. Bronson, Z. Amsden, G. Cabrera, P. Chakka, P. Dimov, H. Ding, J. Ferris, A. Giardullo, S. Kulkarni, H. Li, M. Marchukov, D. Petrov, L. Puzar, Y. J. Song, and V. Venkataramani. TAO: Facebook's distributed data store for the social graph. In *Proc. 2013 USENIX Annual Technical Conference*, pages 49–60, 2013.

[8] A. Carzaniga, D. S. Rosenblum, and A. L. Wolf. Achieving scalability and expressiveness in an internet-scale event notification service. In *Proc. 19th ACM Symposium on Principles of Distributed Computing*, pages 219–227, 2000.

[9] M. Castro, P. Druschel, A.-M. Kermarrec, and A. I. Rowstron. Scribe: A large-scale and decentralized application-level multicast infrastructure. *Selected Areas in Communications, IEEE Journal on*, 20(8):1489–1499, 2002.

[10] A. Demers, J. Gehrke, M. Hong, M. Riedewald, and W. White. Towards expressive publish/subscribe systems. In *Advances in Database Technology-EDBT 2006*, pages 627–644. Springer, 2006.

[11] P. T. Eugster, P. A. Felber, R. Guerraoui, and A.-M. Kermarrec. The many faces of publish/subscribe. *ACM Computing Surveys*, 35(2):114–131, 2003.

[12] Facebook, Inc. Graph search. https://www.facebook.com/about/graphsearch, 2014.

[13] P. Hunt, M. Konar, F. P. Junqueira, and B. Reed. ZooKeeper: Wait-free coordination for internet-scale systems. In *Proc. 2010 USENIX Annual Technical Conference*, 2010.

[14] Iron.io, Inc. IronMQ. http://www.iron.io/mq, 2014.

[15] J. Kreps, N. Narkhede, and J. Rao. Kafka: A distributed messaging system for log processing. In *Proc. 6th ACM Workshop on Networking Meets Databases*, 2011.

[16] Y. Liu and B. Plale. Survey of publish subscribe event systems. Technical Report TR-574, Computer Science Dept., Indiana University, 2003.

[17] S. P. Mahambre, M. K. S.D., and U. Bellur. A taxonomy of QoS-aware, adaptive event-dissemination middleware. *IEEE Internet Computing*, 11(4):35–44, 2007.

[18] MySQL AB. Mysql. http://www.mysql.com/.

[19] R. Nishtala, H. Fugal, S. Grimm, M. Kwiatkowski, H. Lee, H. C. Li, R. McElroy, M. Paleczny, D. Peek, P. Saab, D. Stafford, T. Tung, and V. Venkataramani. Scaling Memcache at Facebook. In *Proc. 10th USENIX Symposium on Networked Systems Design and Implementation*, pages 385–398, 2013.

[20] G. Pape. runit. http://smarden.org/runit/, 2014.

[21] M. Petrovic, I. Burcea, and H.-A. Jacobsen. S-ToPSS: Semantic toronto publish/subscribe system. In *Proc. 29th Conference on Very Large Data Bases*, pages 1101–1104, 2003.

[22] Philotic, Inc. Beanstalk. http://kr.github.io/beanstalkd/, 2014.

[23] Pivotal Software, Inc. RabbitMQ. http://www.rabbitmq.com/, 2014.

[24] M. Platania. *Ordering, Timeliness and Reliability for Publish/Subscribe Systems over WAN*. PhD thesis, Sapienza University of Rome, 2011.

[25] V. Ramasubramanian, R. Peterson, and E. G. Sirer. Corona: A high performance publish-subscribe system for the world wide web. In *Proc. 3th USENIX Symposium on Networked Systems Design and Implementation*, pages 15–28, 2006.

[26] W. Rao, L. Chen, A.-C. Fu, H. Chen, and F. Zou. On efficient content matching in distributed pub-/sub systems. In *INFOCOM 2009, IEEE*, pages 756–764. IEEE, 2009.

[27] K. Shvachko, H. Kuang, S. Radia, and R. Chansler. The hadoop distributed file system. In *Proceedings of the 2010 IEEE 26th Symposium on Mass Storage Systems and Technologies (MSST)*, MSST '10, pages 1–10, Washington, DC, USA, 2010. IEEE Computer Society.

[28] R. Strom, G. Banavar, T. Chandra, M. Kaplan, K. Miller, B. Mukherjee, D. Sturman, and M. Ward. Gryphon: An Information Flow Based Approach to Message Brokering. In *Proc. 9th Symposium on Software Reliability Engineering*, 1998.

[29] TIBCO Software, Inc. TIBCO rendezvous concepts. `https://docs.tibco.com/pub/rendezvous/8.3.1_january_2011/pdf/tib_rv_concepts.pdf`, 2014.

[30] `https://github.com/facebook/rocksdb`. Rocksdb. `http://rocksdb.org/`.

[31] S. Q. Zhuang, B. Y. Zhao, A. D. Joseph, R. H. Katz, and J. D. Kubiatowicz. Bayeux: An architecture for scalable and fault-tolerant wide-area data dissemination. In *Proc. 11th International Workshop on Network and Operating Systems Support for Digital Audio and Video*, NOSSDAV '01, pages 11–20, New York, NY, USA, 2001. ACM.

Flywheel: Google's Data Compression Proxy for the Mobile Web

Victor Agababov* Michael Buettner Victor Chudnovsky Mark Cogan Ben Greenstein
Shane McDaniel Michael Piatek Colin Scott[†] Matt Welsh Bolian Yin
Google, Inc. [†]*UC Berkeley*

Abstract

Mobile devices are increasingly the dominant Internet access technology. Nevertheless, high costs, data caps, and throttling are a source of widespread frustration, and a significant barrier to adoption in emerging markets. This paper presents Flywheel, an HTTP proxy service that extends the life of mobile data plans by compressing responses in-flight between origin servers and client browsers. Flywheel is integrated with the Chrome web browser and reduces the size of proxied web pages by 50% for a median user. We report measurement results from millions of users as well as experience gained during three years of operating and evolving the production service at Google.

1 Introduction

This paper describes our experience building and running a mobile web proxy for millions of users supporting billions of requests per day. In the process of developing and deploying this system, we gained a deep understanding of modern mobile web traffic, the challenges of delivering good performance, and a range of policy issues that informed our design.

We focus on mobile devices because they are fast becoming the dominant mode of Internet access. Trends are clear: in many markets around the world, mobile traffic volume already exceeds desktop [23], and double-digit growth rates are typical [32].

Despite these trends, web content is still predominantly designed for desktop browsers, and as such is inefficient for mobile users. This situation is made worse by the high cost of mobile data. In developed markets, data usage caps are a persistent nuisance, requiring users to track and manage consumption to avoid throttling or overage fees. In emerging markets, data access is often priced per-byte at prohibitive cost, consuming up to 25% of a user's total income [18]. In the face of these costs, supporting the continued growth of the mobile Internet and mobile web browsing in particular is our primary motivation.

Although the number of sites that are tuned for mobile devices is growing, there is still a huge opportunity to save users money by compressing web content via a proxy. This paper describes Flywheel, a proxy service integrated into Chrome for Android and iOS that compresses proxied web content by 58% on average (50% median across users). While proxy optimization is an old idea [15, 22, 29, 39, 41] and the optimizations we apply are known, we have gained insights by studying a modern workload for a service deployed at scale. We describe Flywheel from an operational and design perspective, backed by usage data gained from several years of deployment and millions of active users.

Flywheel's data reduction benefits rely on cooperation between the browser and server infrastructure at Google. For example, Chrome has built-in support for the SPDY [11] protocol and the WebP [12] image compression format. Both improve efficiency, yet are rarely used by website operators because they require cumbersome, browser-specific configuration. Rather than waiting for all sites to adopt best practices, Flywheel applies optimizations automatically and universally, transcoding content on-the-fly at Google servers as it is served.

This paper makes two key contributions. First, our experience with Flywheel has given us a deep understanding of the performance issues with proxying the modern mobile web. Although proxy optimization delivers clear-cut benefits for data reduction, its impact on latency is mixed. Measurements of Flywheel's overall performance demonstrate the expected result: compression improves latency. In practice however we find that Flywheel's impact on latency varies significantly depending on the user population and metric of interest. For example, we find that Flywheel decreases load time of large pages but increases load time for small pages.

Our second contribution is a detailed account of the many design tradeoffs and measurement findings that we encountered in the process of developing and deploying Flywheel. While the idea of an optimizing proxy is conceptually simple, our design has evolved continuously in response to deployment experience. For example, we find that middleboxes within mobile carri-

[*] Authors are listed in alphabetical order.

ers are widespread, and often modify HTTP headers in ways that break naïve proxied connections. While use of HTTPS and SPDY would prevent tampering, always encrypting traffic to the proxy is at odds with features such as parental controls enforced by mobile carriers. Perhaps unsurprisingly, addressing these tussles consumes significant engineering effort. We report on the incidence and variety of these tussles, and map them to a clearer picture of mobile web operation with the hope that future system designs will be informed by the practical concerns we have encountered. As far as we know, we are the first to publish a discussion of these tradeoffs.

2 Background

We built Flywheel in response to the practical stumbling blocks of today's mobile web. Ideally, Flywheel would be unnecessary. Mobile data would be cheap, and content providers would be quick to adopt new technologies. Neither is true today.

Mobile Internet usage is large and growing rapidly. The massive growth of mobile Internet traffic has created a tremendous opportunity for automatic optimization. In Asia and Africa, 38% of web page views are performed on mobile devices as of May 2014, a year-over-year increase exceeding 10% [36]. In North America, mobile page loads are 19% of total traffic volume with 8% growth yearly. In February 2014, research firm comScore reported that time spent using the Internet on mobile devices exceeded desktop PCs for the first time in the United States [23]. These trends match our experience at Google. Mobile is increasingly dominant.

Growth in emerging markets is hampered by cost. Emerging markets are growing faster than developed markets. Year-over-year growth in mobile subscriptions is 26% in developing countries compared to 11.5% in developed countries. In Africa, growth exceeds 40% annually [32]. Despite surging popularity, the high cost of mobile access encumbers usage. One survey of 17 countries in sub-Saharan Africa reports that mobile phone spending was 10-26% of individual income in the lower-75% income bracket [18].

Site operators are slow to adopt new technologies. Adapting websites for mobile involves manual and often complex optimizations, and most sites are poorly equipped to make even simple changes. For example, measurements of Flywheel's workload show that 42% of HTML bytes on the web that would benefit from compression are uncompressed, despite GZip being universally supported in modern web browsers [13]. This is in part because GZip is not enabled by default on most web servers, yet only a single-line change to the server configuration is needed. While hosting-providers and CDNs deal with such configuration issues on the behalf of content providers, the pervasive lack of GZip usage indicates that most content providers still do not employ these services.

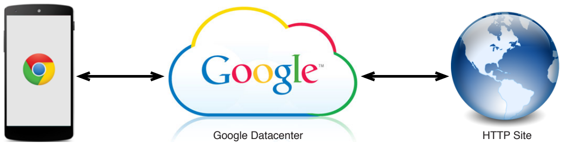


Figure 1: Flywheel sits between devices and origins, automatically optimizing HTTP page loads.

More recent optimizations such as WebP [12] and SPDY [11] have been available for years yet have very low adoption rates. We find that 0.8% of images on the web are encoded in the WebP format, and only 0.9% of sites support SPDY [49].

Users should not have to wait for sites to catch up to best practices. Modern browsers such as Chrome are updated as often as every six weeks, providing a constant stream of new opportunities for optimization that are difficult for web developers to track. Moreover, as mobile devices proliferate, the complexity of optimizing sites to conform to the latest platforms (e.g. high-resolution tablets requiring higher image quality) is a daunting task for all but the most committed site owners.

In sum, it is not surprising that most site owners do not take advantage of all browser- and device-specific optimizations. Just as we do not expect programmers to manually unroll loops, we should not expect site owners to remember to apply an ever-expanding set of optimizations to their sites. We need an optimizing compiler for the web—a service that automatically applies optimizations appropriate for a given platform.

3 Design & Implementation

This section describes Flywheel's design and implementation. The high-level design (depicted in Figure 1) is conceptually simple: Flywheel is an optimizing proxy service. Chrome sends HTTP requests to Flywheel servers running in Google datacenters. These proxy servers fetch, optimize, and serve origin responses. An example optimization is transcoding a large, lossless PNG image into a small, lossy WebP. The remainder of this section describes our goals, the flow of requests and responses, and the optimizations applied in transit. We conclude the section with a discussion of fault tolerance.

3.1 Goals & Constraints

Flywheel's primary goal is to reduce mobile data usage for web traffic. To achieve this goal we must address the practical requirements of integrating with the Chrome browser, used by hundreds of millions of people.

Opt-in. Recognizing that many users are sensitive to the privacy issues of proxying web content through Google's servers, we choose to keep the Flywheel proxy off by default. Users must explicitly enable the service.

Proxy HTTP URLs only. Flywheel applies only to HTTP URLs. HTTPS URLs and page loads from incognito tabs[1] do not use the proxy. While it is technically feasible to proxy through Flywheel in these cases, we are deliberately conservative when given an explicit signal that a request is privacy sensitive.

Maintain transparency for users, network operators, and site owners. Flywheel does not depend on mobile carriers to change their networks or site operators to change their content. Once enabled, Flywheel is transparent to users: websites should look and behave exactly as they would without the proxy in use. This requirement means we must be fairly conservative in terms of the types of optimizations we perform—for example, we cannot modify the DOM of a given page without risking adverse interactions with JavaScript on that page. Further, unavailability of Flywheel service should not prevent users from loading pages.

Comply with standards. All of the optimizations performed by Flywheel must be compliant with modern web standards, including the practical reality of middleboxes that may cache or transform the optimized content. Further, we use standard protocols (SPDY and HTTP) for transferring content in order to ensure that Flywheel can be widely deployed without compatibility issues.

While improving web page load times through Flywheel is desirable, it is not always possible, as we describe later. There is a latency cost to proxying web content through third-party proxies, and although compressing responses tends to reduce load times, this benefit does not always outweigh the increased latency of fetching content through Flywheel servers.

3.2 Client Support: Chrome

Flywheel compression is a feature of the Chrome browser on Android and iOS. Users enable Flywheel in Chrome's settings, which shows a graph of compression over time once enabled.

SPDY (HTTP/2). By default, client connections to Flywheel use the SPDY[2] [11] transport protocol, which multiplexes the transfer of HTTP content over a single TLS connection. SPDY is intended to improve performance as well as security by avoiding the overhead of multiple TCP connections and prioritizing data transfer. For example, HTML and JavaScript are often on the critical path for rendering a page and hence have higher transfer priorities.

For users of Flywheel, SPDY support is universal. Each client application maintains a single SPDY connection to the Flywheel proxy over which multiple HTTP requests are multiplexed. Flywheel in turn translates these to ordinary HTTP transactions with origin servers.

Proxy bypass. A practical reality is that Flywheel cannot proxy all pages on the web. Some sites are simply inaccessible to the Flywheel proxy, such as those behind a corporate intranet or private network. Other sites actively block traffic from Flywheel, for example, due to automated DoS prevention that interprets the volume of traffic from Flywheel IP addresses as an attack.

To avoid unavailability, Flywheel is automatically disabled in these circumstances using a mechanism we call *proxy bypass*. Proxy bypass is implemented using a special HTTP control header that informs the browser to disable the proxy and reload the affected content directly from the origin site. Proxy bypasses are configurable, giving us the ability to disable Flywheel for a set time (e.g., to cover an entire page load) or for just a single URL. This signal also provides us with a convenient load shedding mechanism: we can remotely disable Flywheel for specific clients as needed. We describe the uses of proxy bypass in greater detail in §3.3.5.

HTTP fallback. SPDY is desirable for Flywheel proxy connections due to its performance advantages (§4) and insulation from middlebox interference. However, because SPDY traffic is encrypted, its use can interfere with adult content filtering deployed by mobile carriers, schools, etc. Many mobile carriers also perform selective modification of HTTP request headers from clients in their network, for example to support automatic login to a billing portal site. Although web content filtering can be used as a means of censorship, our goal is not to circumvent such filtering with Flywheel; we wish to be "filter neutral."

We therefore provide a mechanism whereby the connection to the Flywheel proxy can fall back to unencrypted HTTP. This is typically triggered using the proxy bypass mechanism described earlier, although the effect is to switch the proxy connection from SPDY to HTTP, rather than disabling Flywheel entirely.

We also provide a mechanism whereby network operators can disable the use of SPDY Flywheel connections for specific clients in their network [3]. While establishing a SPDY connection, the client makes an unencrypted HTTP request to a well-known URL hosted by Google. If the response contains anything other than an expected string, the client assumes that the URL was blocked by an intermediary and disables use of SPDY for the Flywheel connection. This mechanism is straightforward for carriers to use and allows them to achieve their goals. More complicated approaches that we considered would have required significant integration work between carriers and Google.

In some cases, SPDY must be disabled to avoid triggering bugs in sites. One example we encountered is a

[1] Incognito mode is a Chrome feature that discards cookies, history, and other persistent state for sites visited while it is enabled.

[2] As the HTTP/2 protocol based on SPDY moves to the final stages of standardization, we are migrating from SPDY to HTTP/2.

Technique	Section
HTTP caching	§3.3.4
Multiplexed fetching	§3.3.1
Image transcoding	§3.3.2
GZip compression	§3.3.2
Minification	§3.3.2
Lightweight 404s	§3.3.2
TCP preconnect	§3.3.3
Subresource prefetching	§3.3.3
Header integrity check	§3.3.5
Anomaly detection	§3.3.5

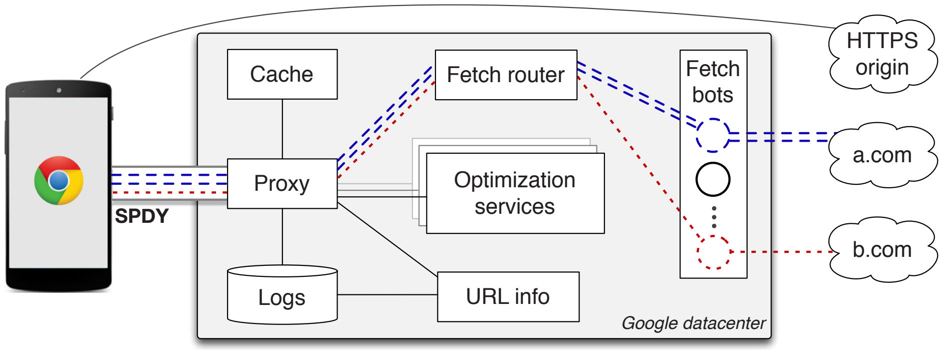


Figure 2: The Flywheel server architecture within a datacenter. Lines indicate bidirectional communication via RPC or HTTP. Each logical component is comprised of replicated, load-balanced tasks for scalability and fault tolerance. The majority of Flywheel code is written in Go, a fact we mention only to dispel any remaining notion that Go is not a robust, production-ready language and runtime environment.

site that uses JavaScript to inspect the response headers of an AJAX request. To improve compression, SPDY sends all headers as lowercase strings, as permitted by the HTTP standard. However, the site's JavaScript code uses a case-sensitive comparison of header names in its logic, leading to an unexpected execution path that ultimately breaks page rendering if SPDY is enabled.

Safe Browsing. Safe Browsing is a feature of Chrome that displays a warning message if a user is about to visit a known phishing or malware site [2]. Because of the overhead of synchronizing Safe Browsing data structures, this feature was disabled for mobile clients and, as of February 2015, is only now being gradually enabled. The main obstacle to its deployment is the tradeoff between bandwidth consumption, power usage, coverage, and timeliness of updates. Ideally, all clients would learn of new bad URLs instantly, but synchronization overhead can be significant, requiring care in managing the tradeoffs. At the Flywheel server, however, many of these tradeoffs do not apply. Flywheel checks all incoming requests against malware and phishing lists, providing an additional layer of protection that is always up-to-date. For requests that match bad URLs, the server signals the client to display a warning.[3]

3.3 Server

The Flywheel server runs in multiple Google datacenters spread across the world. Client connections are directed to a nearby datacenter using DNS-based load balancing; e.g., a client resolving the Flywheel server hostname in Europe is likely to be directed to a datacenter in Europe.

The remainder of this section describes the data reduction and performance optimizations applied at the Flywheel server. Figure 2 provides an overview of our techniques and architecture.

3.3.1 Multiplexed Fetching

The proxy coordinates all aspects of handling a request. The first step is to match incoming requests against URL patterns that should induce a Safe Browsing warning or a proxy bypass. For requests that match, a control response is immediately sent to the client. Otherwise, the request is forwarded via RPC to a separate fetch service that retrieves the resource from the origin.

The fetch service is distinct from the proxy for two reasons. First, a distinct fetch service simplifies management. Many teams at Google need to fetch external websites, and a shared service avoids duplicating subtle logic like rate-limiting and handling untrusted external input. The second benefit to a separate fetch service is improved performance. As shown in Figure 2, the fetch service uses two-level request routing. Request RPCs are load balanced among a pool of fetch routers, which send requests for the same destination to the same fetching bot. Bots are responsible for the actual HTTP transaction with the remote site. Request affinity facilitates TCP connection reuse—requests from multiple users destined for the same origin can be multiplexed over a pool of hot connections rather than having to perform a TCP handshake for each request. Similarly, the fetch service maintains a shared DNS cache, reducing the chance that DNS resolution will delay a request.

3.3.2 Data Reduction

After receiving the HTTP response headers from the fetch service, the proxy makes a decision about how to compress the response based on its content type.[4] These optimizations are straightforward and we describe them briefly. A theme of our experience is that data reduction is the easy part. The bulk of our exposition and engineering effort is dedicated to robustness and performance.

[3] Of course, client checks are still beneficial since Flywheel is not enabled for all users and does not proxy HTTPS and incognito requests.

[4] The proxy respects Cache-Control: No-Transform headers used by origins to inhibit optimization.

Some optimizations are performed by the proxy itself; others are performed by separate pools of processes coordinated via RPC (see Figure 2). Separating optimization from the serving path allows us to load balance and scale services with different workloads independently. For example, an image conversion consumes orders of magnitude more resources than a cache hit, so it makes sense to consolidate image transcoding in a separate service.

Distinct optimization services also improve isolation. For example, because of the subtle bugs often encountered when decoding arbitrary images from the web, we wrap image conversions in a syscall sandbox to guard against vulnerabilities in image decoding libraries. If any optimization fails or a bug causes a crash, the proxy recovers by serving the unmodified response.

Image transcoding. Flywheel transcodes image responses to the WebP format, which provides roughly 30% better compression than JPEG for equivalent visual quality [12]. To further save bytes, we also adjust the WebP quality level according to the device screen size and resolution; e.g. tablets use a higher quality setting than phones. Animated GIF images are transcoded to the animated WebP format. Very rarely, the transcoded WebP image is larger than the original, in which case we serve the original image instead.

Minification. For JavaScript and CSS markup, Flywheel minifies responses by removing unnecessary whitespace and comments. For example, the JavaScript fragment

```
// Issue a warning if the browser
// doesn't support geolocation.
if (!navigator.geolocation) {
  window.alert(
    "Geolocation is not supported.");
}
```

is rewritten (without line breaks) as:

```
if(!navigator.geolocation){window.alert
("Geolocation is not supported.");}
```

GZip. Flywheel compresses all text responses using GZip [26]. This includes CSS, HTML, JavaScript, plain text replies, and HTML. Unlike image optimization, which requires buffering and transcoding the complete response, GZipped responses are streamed to clients. Streaming improves latency, as the browser can begin issuing subresource requests before the HTML download completes.

Lightweight error pages. Many requests from clients result in a 404 error response from the origin, for example, due to a broken link. However, in many cases the 404 error page is not shown to the user. For example, Chrome automatically requests a small preview image called a favicon when navigating to a new site, which often results in a 404. Analyzing Flywheel's workload shows that 88% of page loads result in a 404 error being returned for the favicon request. These error pages can be quite large, averaging 3.2KB—a fair number of "invisible" bytes for each page load lacking a favicon. Flywheel returns a small (68 byte) response body for favicon and apple-touch-icon requests that return a 404 error since the error page is not typically seen by the user.

3.3.3 Preconnect and Prefetch

Preconnect and prefetch are performance optimizations that reduce round trips between the client and origin server. The key observation is that while streaming a response back to the client, the proxy can often predict additional requests that the client will soon make. For example, image, JavaScript and CSS links embedded in HTML will likely be requested after the HTML is delivered. Flywheel parses HTML and CSS responses as they are served in order to discover subresource requests.

Another source of likely subresource requests comes from the URL info service (see Figure 2), which is a database populated by an analysis pipeline that periodically inspects Flywheel traffic logs to determine subresource associations, e.g. requests for `a.com/js` are followed by requests for `b.com/`. This offline analysis complements online parsing of HTML and CSS since it allows Flywheel to learn associations for resources requested by JavaScript executed at the client. We eschew server-side execution of JavaScript because of the comparatively high resource requirements and operational complexity of sandboxing untrusted JavaScript.

Given subresource associations, Flywheel either prefetches the entire object or opens a TCP preconnect to the origin. Which of these is used is determined by a policy intended to balance performance against server overhead. Server overhead comes from wasted preconnects or prefetches that are not used by a subsequent client request. These can arise in case of a client cache hit, a CSS resource for a non-matching media query, or an origin response that is uncacheable. Avoiding these cases would require Flywheel to maintain complete information about the state of the client's cache and cookies. Because of privacy concerns, however, Flywheel does not track or maintain state for individual users, so we have no basis for storing cache entries and cookies.

Flywheel balances the latency benefits of prefetch and preconnect against overhead by issuing a bounded number of prefetches per request for only the CSS, JavaScript, and image references in HTML and only image references in CSS. Preconnects are similarly limited. We track cache utilization and fraction of warm TCP connections to tune these thresholds, a topic we revisit in §4.

3.3.4 HTTP Caching

Flywheel acts as a customized HTTP proxy cache.

Customized entry lookup. Flywheel may store multiple optimized responses for a single URL, for example a transcoded WebP image with two different quality levels—one for phones and another for tablets. Similarly, only some versions of Chrome support WebP animation, so we also need to distinguish cache entries on the basis of supported features. Dispatching logic is shared by both the cache and optimization path in the proxy, so that a cache hit for a particular request corresponds to the appropriate optimized result. The client information is included in each request, e.g., the User-Agent header identifies the Chrome version and device type.

Private external responses. When serving responses over HTTP rather than SPDY, Flywheel must prevent downstream caches from storing optimized results since those caches will not share our custom logic.[5] Downstream proxy caching can break pages, e.g., by serving a transcoded WebP image to a client that does not support the format. To prevent this, we mark all responses transformed by Flywheel as Cache-Control: private, which indicates that the response should not be cached by any downstream proxy but may be cached by the client.

3.3.5 Anomaly Detection

Flywheel employs several mechanisms for improving robustness and availability.

Proxy bypass. Transient unavailability (e.g. network connection errors, software bugs, high server load) may occur along the path between the client, proxy, and origin. In these cases, Flywheel uses the proxy bypass mechanism described earlier to hide such failures from users. Recall that proxy bypass disables the proxy for a short time and causes the affected resources to be loaded directly from the origin site. Proxy bypass is triggered either when an explicit control message is received from the proxy, or when the client detects abnormal conditions. These include:

- *HTTP request loop:* A loop suggests a misconfigured origin or proxy bug. If the loop continues without Flywheel enabled, client-side detection is triggered.
- *Unreachable origin:* DNS or TCP failures at the proxy suggest network-level unavailability, an attempt to access an intranet site, or Google IP ranges being blocked by the origin.
- *Server overload:* The proxy sheds load if needed by issuing bypasses that disable Flywheel at a particular client for several minutes (§4.4).
- *Blacklisted site or resource:* Sites that are known to have correctness problems when fetched via Flywheel are always bypassed, e.g. carrier portals that depend on IP addresses to identify subscribers.
- *Missing control header:* Middleboxes may strip HTTP control headers used by Flywheel if SPDY is disabled. If the proxy observes that such headers are missing from the client request, it sends a bypass to avoid corner cases wherein non-compliant HTTP caches may break page loads.
- *Unproxyable request:* We bypass requests for loopback, .local, or non-fully qualified domains.

Fetch failures. Some fetch errors can be recovered at the server without bypassing, e.g. DNS resolution or TCP connection failures. Simply retrying a fetch often works, recovering what would otherwise have been a bypass.

While retrying a failed fetch often succeeds, it can increase tail latency in the case that an origin is persistently flaky or truly unavailable. To detect these cases, we use an anomaly detection pipeline to automatically detect flaky URLs; i.e., those that have high fetch failure rates. There are thousands of such URLs, making manual blacklisting impractical.

The analysis pipeline runs periodically, inspects traffic logs, and determines URLs that have high fetch failure rates. These URLs are stored in the URL info service, which Flywheel consults upon receiving each request. Flaky URLs are bypassed immediately without waiting for multiple failed retries. To avoid blacklisting a URL forever, Flywheel allows a small fraction of matching requests to proceed to the origin to test if the URL has become available. The analysis pipeline removes an entry from the blacklist provided the failure rate for the URL has dropped sufficiently.

Tamper detection. As described in Section 3.2, Chrome will occasionally fall back to using an HTTP connection to the Flywheel proxy. The need for unencrypted transport is not uncommon; 12% of page loads through Flywheel use HTTP.

Unencrypted transport means that both the Flywheel client and server need to be robust to modifications by third-party middleboxes. For example, we have found that middleboxes may strip our control headers on either the request or response path. Or, they may ignore directives in the Cache-Control header and serve cached Flywheel responses to other users. They may also optimize and cache responses independently of Flywheel. Our experience echoes other studies: transparent middleboxes are common [45, 56, 57].

To provide robustness to middleboxes, Flywheel is defensive at both the client and server, bypassing the proxy upon observing behavior indicating middlebox tampering. Examples include TLS certification validation failures (typical of captive portals) or missing

[5] Responses delivered via SPDY cannot be cached by intermediate proxy caches because of TLS encryption.

Flywheel headers (typical of transparent caches). In practice, checking for these cases has been sufficient to avoid bugs. An overly conservative policy, however, risks eroding data savings, since we need not disable Flywheel in all circumstances. For example, while non-standard middlebox caching risks breaking pages (e.g. serving Flywheel responses to non-Flywheel users), HTTP-compliant caching is mostly benign (e.g. at worst serving images optimized for tablets to non-tablets). Similarly, Flywheel should be disabled in the presence of a captive portal, but only until the user completes the sign-in process. We continue to refine our bypass policies, and this refinement will continue as the behavior of middleboxes evolves.

4 Evaluation

Our evaluation of Flywheel is grounded in measurements and analysis of our production workload comprising millions of users and billions of queries per day. We focus on data reduction, performance, and fault tolerance.

Data in this section is drawn from two sources: (1) Flywheel server traffic logs, which provide fine-grained records of each request, and (2) Chrome *user metrics* reports, which are aggregated distributions of metrics from Chrome users who opt to anonymously share such data with Google.[6]

4.1 Workload

Since Flywheel is an optional feature of Chrome, only a fraction of users have it enabled. Adoption tends to be higher (14-19%, versus 9% worldwide) in emerging countries such as Brazil where mobile data is costly. Although we do not know whether Flywheel has changed user behavior as we hoped in emerging countries, higher adoption rates indicate a perceived benefit.

Access network. Segmented by access network, we find that 78% of page loads are transferred via WiFi, 11% via 3G, 9% via 4G/LTE, and 1% via 2G. Unsurprisingly, the majority of browsing using Flywheel is via WiFi, since the proxy is enabled regardless of the network type the device is using. While the browser could disable Flywheel on WiFi networks, this would eliminate other benefits of Flywheel such as safe browsing. WiFi is prevalent in terms of traffic volume for several reasons: first, it tends to be faster, so users on WiFi generate more page views in less time. A second reason is that tablets are more likely to use WiFi than cellular data.

Traffic mix. Recall that Flywheel does not receive all traffic from the client; HTTPS and incognito page loads are not proxied. For the 28 day period from August 11 through September 8, 2014, we see that 37% of the total bytes downloaded by users (after optimization) with Flywheel enabled are received from the proxy. In comparison, 50% of total received bytes are over HTTPS,[7] and the remainder are incognito requests, bypassed URLs and protocols other than HTTP/HTTPS (e.g. FTP). A notable consequence of this traffic mix is that because our servers only observe a fraction of web traffic, the data reduction observed at Flywheel servers translates into a smaller overall reduction observed at clients.

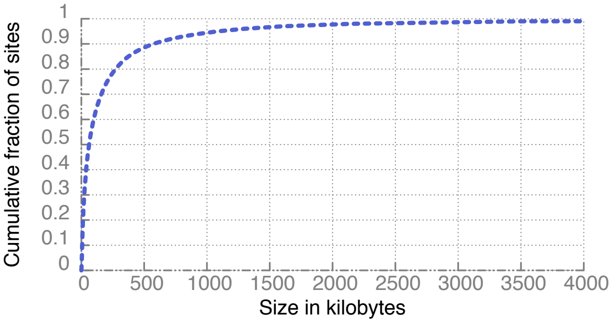


Figure 3: The cumulative distribution of web page size (summation of object sizes) in bytes.

Page footprints. Figure 3 shows the distribution of web page sizes observed in our workload, calculated as the sum of the origin response body bytes for all resources on the page. This distribution is dominated by a small number of larger sites. The median value of 63 KB is dwarfed by the 95th percentile exceeding 1 MB. The tendency for total data *volume* to be dominated by a small number of page loads but the typical page load *time* to be dominated by a large number of very small pages has implications for the latency impact of proxy optimization, a topic we discuss in §4.3.

Video. Flywheel does not currently compress video content, for two reasons. First, most mobile video content is downloaded by native apps rather than the browser. Second, video content embedded in web pages is loaded not by Chrome but by a separate Android process in most cases; hence, video does not pass through the Chrome network stack and cannot be proxied by Flywheel. However, preliminary work on video transcoding using the WebM format suggests that we can expect to achieve 40% data reduction without changing the frame rate or resolution.

4.2 Data reduction

Overall. Excluding request and response headers, Flywheel reduces the size of proxied content by 58% on average. Reduction is computed as the difference between

[6] Chrome users can see a complete list by navigating to about:histograms.

[7] Just 33% of total bytes were received over HTTPS 9 months prior—aggregated between March 11th and April 8th—representing a noteworthy 17 percentage point increase in HTTPS adoption over 9 months.

Type	% of Bytes	Savings	Share of Benefit
Images	74.12%	66.40%	85%
HTML	9.64%	38.43%	6%
JavaScript	9.10%	41.09%	6%
CSS	1.81%	52.10%	2%
Plain text	0.64%	20.49%	.2%
Fonts	0.37%	9.33%	.1%
Other	4.32%	7.76%	1%

Table 1: Resource types and data reduction.

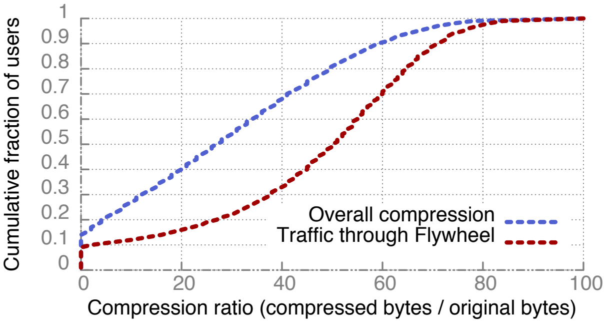


Figure 4: Distribution of overall data reduction across users. The overall reduction is lower than that through Flywheel because we do not proxy HTTPS or incognito traffic.

total incoming and total outgoing bytes at the proxy (excluding bytes served out of cache) divided by total incoming bytes. Table 1 segments traffic and data reduction by content type for a day-long period in August 2014. The 'Other' category includes all other content types, missing content types, and non-200 responses. Note that 'Other' does not include large file transfers: because these are typically binary files that compress poorly, Flywheel sends a proxy bypass upon receiving a request for a large response.

The majority of data reduction benefit comes from transcoding images to WebP, which reduces the size of images by 66% on average. Much of the remaining data reduction benefits come from GZipping uncompressed responses, with larger benefits for CSS and JavaScript due to syntactic minification.

Overall reduction. The data reduction at the server is an upper bound for reduction observed by clients. Recall that HTTPS and incognito page loads, as well as bypassed traffic, are not compressed by Flywheel. Figure 4 quantifies the difference. Across users, the median data reduction for all traffic is 27%, compared to 50% for traffic proxied through Flywheel.

WebP quality. Because images dominate our workload, the aggressiveness of our WebP encoding has a significant influence on our overall data reduction. Our goal is to achieve as much reduction as possible without affecting the perceived quality of the image. To tune the quality, we used the structural similarity index metric (SSIM) [53] to compare the visual similarity of 1500 original vs. compressed images (drawn from a set of 100 popular curated URLs) for different WebP encoder quality values. We picked initial quality values by choosing the knee of the SSIM vs. quality curve.

Of course, the ideal visual quality metric is actual user perception, and there is no substitute for experience. Our experimentation with internal Google users before launch lead us to transcode images at quality 50 for phones and 70 for tablets, which roughly corresponds to an SSIM threshold of ~0.85 and ~0.9, respectively. Prior to tuning, we had received several complaints from internal testers regarding visual artifacts; we have no known reports of users complaining about the current settings.

Lightweight error responses. Flywheel sends a small (68 byte) response body for 404 errors returned for favicon and apple-touch-icon requests. Despite the fact that 404 responses for these images constitute only 0.07% of requests, the full error pages would account for 2% of the total data consumed by Flywheel users. The average 404 page for an apple-touch-icon is 3.3KB, and two such requests are made for every page load. Our lightweight 404 responses eliminate nearly all of this overhead.

Redundancy elimination. Our workload provides a scaffold for evaluating potential data reduction optimizations. Sometimes, this yields a negative result. We conclude our evaluation of data reduction with two such examples.

Many websites do a poor job of setting caching parameters, e.g. using time-based expiration rather than content-based ETags. Others mark resources that do not change for months as uncacheable. The result is that clients unnecessarily download resources that would otherwise be in their cache.

Flywheel could improve data reduction by fixing these configuration errors. For example, we could add a content-based ETag to all responses lacking one, and verify that the origin content is unchanged upon each revalidation request sent by the client.

We evaluate the potential benefits of redundancy elimination using trace replay: we measure the possible increase in data reduction from eliminating all redundant transfers across all client sessions, where a session is defined as the duration between browser restart events.[8] We define a redundant transfer as two response bodies with exactly matching contents (we discuss partial matches next).

The result of redundancy elimination is a modest improvement in data reduction. Overall, 11.5% of the bytes served are redundant after data reduction. Restricting

[8] This typically extends beyond a single foreground session, since Android does not prune tasks except under memory pressure.

consideration to only JavaScript, CSS, and images reduces the benefit to 7.8%. Given this data, we concluded that this opportunity was not worth prioritizing. The assumption of complete redundancy elimination is optimistic, and impact of data reduction by the server would be reduced by the limited fraction of traffic handled by Flywheel at the client. We may return to this technique, but not before pursuing simpler and more fruitful optimizations such as video compression.

Delta Encoding. Most compression techniques focus on reducing data usage for a single response object. If an object is requested multiple times however, the origin may have only made small modifications to the object between the first and subsequent request.

Chrome supports a delta encoding technique known as Shared Dictionary Compression over HTTP (SDCH) designed to leverage such cross-payload redundancy [17] by only sending the deltas between modified objects rather than the whole object. We modified Flywheel to support server-side SDCH functionality (e.g. dictionary construction) on behalf of origin servers. We then duplicated a fraction of user traffic, applied SDCH to the duplicated traffic, and measured what the data reduction would have been if we had served the SDCH responses to users. We found that SDCH only improved data reduction for HTML and plain text objects from ~35% to ~41%, equivalent to less than 1% improvement in overall data savings. We therefore opted to not deploy SDCH.

4.3 Performance

We next examine Flywheel's impact on latency. Compared to data reduction, evaluating performance is significantly more complex. The results are mixed: Flywheel improves some performance metrics and degrades others. These results reflect a tradeoff between compression, performance, and operating environment. Table 2 summarizes the performance data underlying these results, which we describe in detail below.

Methodology. Our evaluation answers two main questions: (1) Does Flywheel improve latency compared to loading pages directly? And, (2) how effective are the server-side mechanisms used to improve latency?

We use Chrome user metrics reports (described earlier) to gather client-reported data on page load time (PLT), time to first paint, time to first byte (TTFB), and so on. These anonymous reports are aggregated and can be sliced by a variety of properties, e.g. the client version, device type, country, and network connection. We use server-side logs to measure the effectiveness of performance optimizations such as multiplexed fetching, preconnect, and prefetch. Clients are unaware as to whether or not these optimizations are applied, so their use is not reflected in user reports. Instead, we evaluate these using server traffic logs.

For both client-side and server-side measurements, all comparisons are made relative to a *holdback experiment*, a random sampling of 1% of users for which the proxy is silently disabled, despite the feature being turned on by the user. A holdback group is essential for eliminating sampling bias. For example, comparing the latency observed by users with Flywheel on and off suggests a significant increase in page load time due to Flywheel. However, the holdback experiment shows that the typical page load time of a user who enables Flywheel is higher than the overall population of Chrome users. In retrospect, this is unsurprising—users are more likely to enable Flywheel if they are bandwidth-limited, and Flywheel adoption rates are highest in countries with comparatively high page load times.

Flywheel reduces page load time when pages are large. For most users and most page loads, Flywheel increases page load time. This is reflected in Table 2; cf. rows for 'Holdback' and 'Flywheel'. For the majority of page loads, the increase is modest: the median value increases by 6%. The benefits of compression appear in the tail of the distribution, with the PLT reduction at the 95th percentile being 5%. We attribute this to our production workload: data reduction improves latency when pages are large, and as shown in Figure 3, the distribution of page load sizes is heavily skewed.

Flywheel's performance benefit arises from a combination of individual mechanisms. For example, we find that SPDY provides a median 4% reduction in page load time relative to proxying via HTTPS. Data reduction improves latency, but only for the minority of large pages; e.g. disabling all data reduction optimizations increases median page load time through the proxy by just 2%, but the 95th percentile PLT increases by 7%. On the whole, the contribution of individual mechanisms varies significantly based on characteristics of clients and sites.

Flywheel increases time to first paint. Page load time is an upper bound on latency. But, long before a page is loaded fully, it may display useful content and become interactive. Moreover, displaying a partially rendered page increases perceived responsiveness even if the overall load time is unchanged. To capture a lower bound on page load performance, we consider time to first paint; i.e., the time after loading begins when the browser has enough information to begin painting pixels.

As shown in Table 2, Flywheel increases median time to first paint by 10%. This increase is modest, and drops off in the tail of the distribution. A probable cause of this increase is a corresponding inflation of time to first byte; i.e., the delay between sending the first request in a page load and receiving the first byte of the response. Flywheel increases median TTFB by 40%. Unlike time to first paint, we observe inflated TTFB at all quantiles

Flywheel configuration	*Median*		*70th*		*80th*		*90th*		*95th*		*99th*	
Page load time quantiles (milliseconds)												
Holdback	2075		3682		5377		9222		14606		39380	
Flywheel	2207	6.4%	3776	2.6%	5374	-0.1%	8951	-2.9%	13889	-4.9%	36130	-8.3%
Holdback (Beta)	2123		3650		5151		8550		13192		32650	
Images only (Beta)	2047	-3.6%	3447	-5.6%	4944	-4.0%	8214	-3.9%	12476	-5.4%	31650	-3.1%
Japan, page load time quantiles (milliseconds)												
Holdback	1355		2289		3133		4939		7211		14926	
Flywheel	1674	23.5%	2715	18.6%	3647	16.4%	5502	11.4%	7797	8.1%	15927	6.7%
Time to first byte quantiles per pageload (milliseconds)												
Holdback	185		360		547		999		1688		5064	
Flywheel	259	40.0%	485	34.7%	687	25.6%	1164	16.5%	1903	12.7%	5808	14.7%
Time to first paint quantiles per pageload (milliseconds)												
Holdback	803		1429		2084		3493		5650		19233	
Flywheel	888	10.6%	1547	8.3%	2194	5.3%	3581	2.5%	5723	1.3%	20374	5.9%

Table 2: Page load time for various Flywheel configurations. Flywheel improves page load time only when pages are large and users are close to a Google data center. All percentages are given relative to the baseline holdback measurement. 'Holdback' refers to a random sampling of 1% of users with Flywheel enabled for whom we disable Flywheel for the browsing session. 'Images only' refers to an experimental configuration wherein only images are proxied through Flywheel. This experiment applies only to Android Chrome Beta users.

Flywheel's latency improvement is not universal. We attribute TTFB inflation primarily to the geographic distribution of Flywheel users. Many users are further from a Google data center than typical web servers, resulting in longer round trips. The extent of TTFB inflation and its relationship to overall latency depends on many factors: latency to Google, from Google to the site, from the user to the site, and the overall benefits of data reduction. Our overall performance data shows that more often than not, the balance of these tradeoffs increases latency.

Usage in Japan provides a concrete example. Flywheel increases median page load time by 23.5% relative to holdback users in Japan, with smaller but still significant increases in the tail. How does this relate to the tradeoffs described above? First, page loads in Japan tend to be fast—34% lower than the overall holdback median. The TTFB inflation is similarly high, but the faster page load time means that the overhead of indirection through Google is proportionally larger. Typical network capacity is also higher in Japan, reducing the benefits of data reduction as round trips represent a larger fraction of overall page load time. While Japan is an extreme case, the overall theme remains: Flywheel's performance benefits are not universal and depend on the interaction of many factors.

At the server, we can further refine the breakdown of TTFB inflation. For Flywheel page loads in the United States over WiFi, for example, median TTFB inflation is 90 milliseconds, of which 60 ms is RTT to Google, 20 ms is RTT from Google to the origin site, and 10 ms is internal routing within Google's network and processing overhead. The precise breakdown of overheads varies by client population, but the dominant factor is typically overhead to reach the nearest Google data center.

Proxying only images improves latency for small pages at the expense of large pages. Given widespread TTFB inflation, the tradeoff between latency and compression is straightforward: Flywheel improves performance when the latency benefit of data reduction outweighs the latency cost of indirect fetching through Google. Trading off data reduction for performance thus requires some notion of *selective proxying*; i.e., sending only some resource loads through Flywheel.

Recall that the majority of Flywheel's data reduction comes from transcoding images to WebP. 74% of bytes passing through Flywheel are images, and 85% of our overall data reduction benefit over a typical day comes from image transcoding (Table 1). This data suggests that proxying requests for images only is likely to eliminate most of Flywheel's latency overhead while retaining most of its data reduction benefit. Implementing this on the client is straightforward: based on surrounding markup, Chrome typically has an expectation of content type; e.g., requests originating from an `<img>` HTML tag are likely to return an image.

The 'Images only' rows in Table 2 show the results of applying this policy as an experiment for Chrome beta users. The results match our intuition. Median page load time is reduced relative to proxying all content. Data reduction is diminished only slightly. On the flip side, the reductions in page load time for the majority of small page loads come at the expense of larger pages. The 99th percentile reduction in PLT from proxying only images is 3% compared with 8% when proxying all content.

Preconnect and prefetch provide modest benefits. We find that although preconnect and prefetch have non-negligible effects on first order metrics (connection reuse and cache hit ratio), they only provide modest 1-2% reductions in median page load time overall. Like other performance metrics, the benefits vary depending on how the data is sliced. The benefits tend to be greater for users with relatively high TTFB inflation. For example, prefetch and preconnect each provide a 2% reduction in PLT for page loads from Japan.

The benefits of preconnect and prefetch are limited by the natural tendency for connection reuse and object caching in our workload. Even without TCP preconnect, for example, 73% of requests from Flywheel are issued over an existing TCP connection. Enabling preconnect increases this fraction to 80%. Similarly, subresource prefetching increases Flywheel's HTTP cache hit rate by 10 percentage points, from 22% to 32%.

Because of their overhead and limited benefits, we have not deployed preconnect or prefetch beyond small experiments because of concerns about the overhead they would impose on origin sites. For example, redundant prefetches increase the number of fetches to origins by 18%. Redundant fetches are caused by two main factors: (1) the prefetched response is already cached at the client, so it will not be requested; or (2) the prefetched response is not cacheable, so we cannot safely use it to satisfy a subsequent client request. We continue to refine our logic for issuing speculative connections and prefetches in an attempt to reduce overhead.

4.4 Fault Tolerance

Our goal is for all Flywheel failures to be transparent to users. If the client cannot contact the Flywheel proxy, or if the proxy cannot fetch and optimize a given URL, our proxy bypass mechanism recovers by transparently requesting resources from the origin server directly (§3.2). Below, we report on the prevalence of bypassed requests, their causes, and mechanisms for improving the precision of proxy bypass.

Bypass causes. A request may be bypassed before it is sent to Flywheel, by Flywheel before it is sent to the origin, or by Flywheel after observing the origin response. We consider each of these cases in turn.

Client-side bypasses are rare, but typically occur due to failure to connect to the proxy. Server-side, 0.89% of requests received by Flywheel result in a bypass being sent to the client. The largest fraction of bypasses (38%) are caused by origin response codes, e.g. 429 indicating a rejected request. Another large fraction are due to fetch errors (28%), e.g. when Flywheel cannot establish a connection to the origin. This could be because the site is down, blocking traffic from the proxy, or because the site is on an intranet or local network not reachable by the proxy. We bypass audio and video files (19% of bypasses) as Flywheel does not currently transcode these response types, as well as large file downloads (0.3%) where we are not able to achieve sufficient data reduction to merit the processing overhead. Requests automatically flagged as problematic by our anomaly detection pipeline constitute 8% of bypasses. We also issue bypasses for blacklisted URLs (5%) for sites with compatibility issues, carrier dependencies, or legal constraints. The remaining 0.25% of bypasses are caused by load shedding; i.e., if an individual Flywheel server becomes unusually slow and accumulates too many in-flight requests, it sends a bypass response.

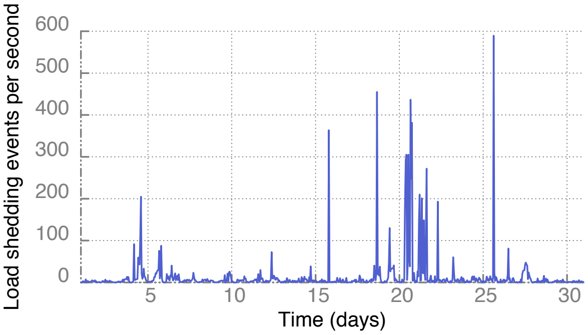


Figure 5: A month-long trace of load shedding events. Transient unavailability is common and is typically resolved without manual intervention.

Load shedding acts as a form of back pressure that provides recovery for many causes of unavailability: congestion, origin slowness, attack traffic, configuration errors, and so on. These events occur frequently, and a stop-gap is essential for smooth operation. A simple load shedding policy of bounding in-flight requests has worked well for us so far. Figure 5 shows a trace of load shedding events collected over one month. Events are typically bursty and short-lived. Crucially, automatic load shedding means that most transient production issues do not require manual intervention for recovery.

Mitigating fetch errors. More than half of bypassed requests are due to fetch errors. Two mechanisms reduce the impact of these errors. First, Flywheel retries failed fetches. Nearly a third of failed fetches succeed after a single retry, and roughly half succeed within 5 retries.

Retrying fetches can impose significant delay if the site is still unreachable after multiple tries. The median latency for fetch errors is ~1 second, with the 90th percentile exceeding 200 seconds. Most fetches that fail after 5 retries are due to DNS lookup or fetch timeouts.

To deal with this, the second mitigation mechanism involves analyzing the server traffic logs periodically to identify URLs that exhibit a high failure rate. If the majority of fetches to a given URL fail, we flag that URL as flaky and push a blacklisting rule into the URL info service to send bypasses for requests to that URL.

Results show that this technique eliminates ~1/3 of all fetch errors. The pipeline achieves a low false positive rate; for 90% of the URLs classified as flaky, at least 70% of the fetches would have failed if they had not been bypassed preemptively. Moreover, a third of bypassed requests would have resulted in a timeout exceeding one second if not bypassed preemptively. Because flaky URLs constitute a small fraction of overall traffic, correcting these errors has limited impact on aggregate page load time. However, exceptionally slow page loads tend to be particularly unpleasant for users, leading us to focus on reducing their impact.

Tamper detection. Our tamper detection mechanisms track cases of middleboxes modifying our HTTP traffic. While we do observe cases of benign tampering, we find that obstructive tampering is rare. For example, over the period of a week, 45 mobile carriers modified our content length header at least once, and 115 carriers appended an extra Via header (indicating the presence of an additional proxy). However, these cases do not significantly hinder user experience; we have only dealt with obstructive tampering on a handful of occasions.

5 Related Work

Optimizing proxy services have received significant attention in the literature, and this work informs our design. This paper differs in two main ways. First, our environment is unique; we focus on the co-design of a modern mobile browser, operating system, and proxy infrastructure. The second difference is scale; we report operational and performance results from millions of users spread across the globe. As far as we know we are the first to report on the incidence and variety of issues encountered by a proxy of this kind.

Web proxies. In the late 90's researchers investigated proxies for improving HTTP performance [22, 39, 41] and accommodating mobile devices [19,29–31,40]. We revisit these ideas in the context of modern optimizations, client platforms, and workloads.

User studies. Others studied the effects of data pricing [21,46], performance [43], and page layout [58] on user behavior. This work reinforces our motivation.

Performance optimizations. At the proxy, we employ known proxy optimizations such as prefetch [16, 35, 41]. By virtue of building on Google infrastructure, we also benefit from transport-level performance optimizations [28, 44]. We do not apply more aggressive optimizations such as 'whole-page' content rewriting [20,34,38] or client offload [48,52], since in our experience even simple changes like CSS import flattening [5] can break some web pages, and our goal is full compatibility with existing pages.

Other work has focused on evaluating existing performance optimizations [16, 24, 27, 47, 50, 51]. Our measurements are derived from a large scale production environment with real user traffic.

Data reduction optimizations. Data reduction techniques beyond those we employ include WiFi offload [14] and differential caching [17,33,37,42,54,59]. Since differential caching only applies to text resources its effect on overall data reduction are limited compared to optimization of images and video, as we quantified in our evaluation of SDCH [17].

Alternate designs. VPN-based compression [4,6,7] offers an alternative to HTTP interposition. The main advantage of VPN-interposition is ubiquity: all traffic can be optimized without modifying applications. But, interposition at the level of raw packets does not lend itself to transport optimizations like SPDY or application-specific mechanisms like proxy bypass, which depends on the client reissuing requests. Flywheel instead integrates with Chrome, which allows us to retain the protocol information required for flexible failure recovery.

Transparent web proxies, which are deployed by many carriers today [25,55–57], present another design option. The main benefit of in-network optimization is that it requires no client modifications whatsoever. But, as with VPNs, interposing at the network level limits options for optimization and failure recovery, and transparent proxies are applicable only within a particular network.

The proxy service with the closest design to ours is Opera Turbo [9]. Although Opera has not published the details of their optimizations or operation, we performed a point comparison of Flywheel and Turbo's data reduction gains, and found that Flywheel provides comparable data reduction.

Other mobile browsers [1, 8, 10] feature more aggressive optimization based on server-side transcoding of entire pages; e.g. Opera Mini rewrites pages into a proprietary format optimized for mobile called OBML [8], and offloads some JavaScript execution to servers rather than clients. While whole-page transcoding can significantly improve data reduction, pages that rely heavily on JavaScript or modern web platform features are often broken by the translation; e.g. touch events are unsupported by Opera Mini [8]. Maintaining an alternative execution environment to support whole-page transcoding is not feasible for Flywheel given our design goal of remaining fully compatible with the modern mobile web.

6 Summary

We have presented Flywheel, a data reduction proxy service that provides an average 58% byte size reduction of HTTP content for millions of users worldwide. Flywheel has been in production use for several years, providing experience regarding the complexity and tradeoffs of operating a data reduction proxy at Internet scale. We find that data reduction is the easy part. The practical realities of operating with geodiverse users, transient failures, and unpredictable middleboxes consume most of our effort, and we report these tradeoffs in the hope of informing future designs.

References

[1] Amazon Silk. http://s3.amazonaws.com/awsdocs/AmazonSilk/latest/silk-dg.pdf.

[2] Chrome SafeBrowse. http://www.google.com/transparencyreport/safebrowsing/.

[3] Data Compresion Proxy Canary URL. https://support.google.com/chrome/answer/3517349?hl=en.

[4] Microsoft Data Sense. http://www.windowsphone.com/en-us/how-to/wp8/connectivity/use-data-sense-to-manage-data-usage.

[5] mod pagespeed. https://developers.google.com/speed/pagespeed/module.

[6] Onavo. http://www.onavo.com/.

[7] Opera Max. http://www.operasoftware.com/products/opera-max.

[8] Opera Mini and Javascript. https://dev.opera.com/articles/opera-mini-and-javascript/.

[9] Opera Turbo. http://www.opera.com/turbo.

[10] Skyfire - A Cloud Based Mobile Optimization Browser. http://www.skyfire.com/operator-solutions/whitepapers.

[11] SPDY Whitepaper. http://www.chromium.org/spdy/spdy-whitepaper.

[12] WebP: A New Image Format For The Web. http://developers.google.com/speed/webp/.

[13] Which Browsers Handle ‘Content-Encoding: gzip‘? http://webmasters.stackexchange.com/questions/22217/.

[14] A. Balasubramanian, R. Mahajan, and A. Venkataramani. Augmenting Mobile 3G with WiFi. MobiSys ’10.

[15] H. Bharadvaj, A. Joshi, and S. Auephanwiriyakul. An Active Transcoding Proxy to Support Mobile Web Access. Reliable Distributed Systems ’98.

[16] C. Bouras, A. Konidaris, and D. Kostoulas. Predictive Prefetching on the Web and its Potential Impact in the Wide Area. WWW ’04.

[17] J. Butler, W.-H. Lee, B. McQuade, and K. Mixter. A Proposal for Shared Dictionary Compression over HTTP. http://lists.w3.org/Archives/Public/ietf-http-wg/2008JulSep/att-0441/Shared_Dictionary_Compression_over_HTTP.pdf.

[18] A. Chabossou, C. Stork, M. Stork, and P. Zahonogo. Mobile Telephony Access and Usage in Africa. African Journal of Information and Communication ’08.

[19] S. Chandra, C. S. Ellis, and A. Vahdat. Application-Level Differentiated Multimedia Web Services using Quality Aware Transcoding. IEEE Selected Areas in Communications ’00.

[20] S. Chava, R. Ennaji, J. Chen, and L. Subramanian. Cost-Aware Mobile Web Browsing. IEEE Pervasive Computing ’12.

[21] M. Chetty, R. Banks, A. Brush, J. Donner, and R. Grinter. You’re Capped: Understanding the Effects of Bandwidth Caps on Broadband Use in the Home. CHI ’12.

[22] C.-H. Chi, J. Deng, and Y.-H. Lim. Compression Proxy Server: Design and Implementation. USITS ’99.

[23] CNN. Mobile Apps Overtake PC Internet Usage in U.S. http://money.cnn.com/2014/02/28/technology/mobile/mobile-apps-internet/.

[24] B. de la Ossa, J. A. Gil, J. Sahuquillo, and A. Pont. Web Prefetch Performance Evaluation in a Real Environment. IFIP ’07.

[25] G. Detal, B. Hesmans, O. Bonaventure, Y. Vanaubel, and B. Donnet. Revealing Middlebox Interference with Tracebox. IMC ’13.

[26] P. Deutsch. GZIP File Format Specification, May ’96. RFC 1952.

[27] J. Erman, V. Gopalakrishnan, R. Jana, and K. Ramakrishnan. Towards a SPDY’ier Mobile Web? CoNEXT ’13.

[28] T. Flach, N. Dukkipati, A. Terzis, B. Raghavan, N. Cardwell, Y. Cheng, A. Jain, S. Hao, E. Katz-Bassett, and R. Govindan. Reducing Web Latency: The Virtue of Gentle Aggression. SIGCOMM ’13.

[29] A. Fox, S. D. Gribble, Y. Chawathe, and E. A. Brewer. Adapting to Network and Client Variation Using Infrastructural Proxies: Lessons and Perspectives. IEEE Personal Communications ’98.

[30] R. Han, P. Bhagwat, R. LaMaire, T. Mummert, V. Perret, and J. Rubas. Dynamic Adaptation in an Image Transcoding Proxy for Mobile Web Browsing. IEEE Personal Communications ’98.

[31] B. C. Housel and D. B. Lindquist. WebExpress: A System for Optimizing Web Browsing in a Wireless Environment. MobiCom '96.

[32] International Telecommunications Union. The World in 2014: ICT Facts and Figures. http://www.itu.int/en/ITU-D/Statistics/Documents/facts/ICTFactsFigures2014-e.pdf.

[33] U. Irmak and T. Suel. Hierarchical Substring Caching for Efficient Content Distribution to Low-Bandwidth Clients. WWW '05.

[34] B. Livshits and E. Kiciman. Doloto: Code Splitting for Network-Bound Web 2.0 Applications. SIGSOFT/FSE '08.

[35] D. Lymberopoulos, O. Riva, K. Strauss, A. Mittal, and A. Ntoulas. PocketWeb: Instant Web Browsing for Mobile Devices. SIGARCH '12.

[36] M. Meeker. Internet Trends 2014. http://pathwaypr.com/must-read-mark-meekers-2014-internet-trends.

[37] L. A. Meyerovich and R. Bodik. Fast and Parallel Webpage Layout. WWW '10.

[38] J. Mickens. Silo: Exploiting JavaScript and DOM Storage for Faster Page Loads. WebApps '10.

[39] J. C. Mogul, F. Douglis, A. Feldmann, and B. Krishnamurthy. Potential Benefits of Delta Encoding and Data Compression for HTTP. SIGCOMM '97.

[40] B. D. Noble and M. Satyanarayanan. Experience with Adaptive Mobile Applications in Odyssey. Mobile Networks and Applications '99.

[41] V. N. Padmanabhan and J. C. Mogul. Using Predictive Prefetching to Improve World Wide Web Latency. SIGCOMM '96.

[42] K. Park, S. Ihm, M. Bowman, and V. S. Pai. Supporting Practical Content-Addressable Caching with CZIP Compression. ATC '07.

[43] A. Patro, S. Rayanchu, M. Griepentrog, Y. Ma, and S. Banerjee. Capturing Mobile Experience in the Wild: a Tale of Two Apps. ENET '13.

[44] S. Radhakrishnan, Y. Cheng, J. Chu, A. Jain, and B. Raghavan. TCP Fast Open. CoNEXT '11.

[45] C. Reis, S. D. Gribble, T. Kohno, and N. C. Weaver. Detecting In-Flight Page Changes with Web Tripwires. NSDI '08.

[46] N. Sambasivan, P. Lee, G. Hecht, P. M. Aoki, M.-I. Carrera, J. Chen, D. P. Cohn, P. Kruskall, E. Wetchler, M. Youssefmir, et al. Chale, How Much It Cost to Browse?: Results From a Mobile Data Price Transparency Trial in Ghana. ICTD '13.

[47] S. Savage, T. Anderson, A. Aggarwal, D. Becker, N. Cardwell, A. Collins, E. Hoffman, J. Snell, A. Vahdat, G. Voelker, et al. Detour: Informed Internet Routing and Transport. IEEE Micro '99.

[48] A. Sivakumar, V. Gopalakrishnan, S. Lee, and S. Rao. Cloud is Not a Silver Bullet: A Case Study of Cloud-based Mobile Browsing. HotMobile '14.

[49] W3Techs. Usage of SPDY for Websites. http://w3techs.com/technologies/details/ce-spdy/all/all.

[50] X. S. Wang, A. Balasubramanian, A. Krishnamurthy, and D. Wetherall. Demystify Page Load Performance with WProf. NSDI '13.

[51] X. S. Wang, A. Balasubramanian, A. Krishnamurthy, and D. Wetherall. How speedy is SPDY? NSDI '14.

[52] X. S. Wang, H. Shen, and D. Wetherall. Accelerating the Mobile Web with Selective Offloading. MCC '13.

[53] Z. Wang, A. C. Bovik, H. R. Sheikh, and E. P. Simoncelli. Image Quality Assessment: From Error Visibility to Structural Similarity. IEEE Image Processing '04.

[54] Z. Wang, F. X. Lin, L. Zhong, and M. Chishtie. How far can client-only solutions go for mobile browser speed? WWW '12.

[55] Z. Wang, Z. Qian, Q. Xu, Z. Mao, and M. Zhang. An untold story of middleboxes in cellular networks. SIGCOMM '11.

[56] N. Weaver, C. Kreibich, M. Dam, and V. Paxson. Here Be Web Proxies. PAM '14.

[57] X. Xu, Y. Jiang, E. Katz-Bassett, D. Choffnes, and R. Govindan. Investigating Transparent Web Proxies in Cellular Networks. Technical Report 007-90818, University of Southern California.

[58] D. Zhang. Web Content Adaptation for Mobile Handheld Devices. CACM '07.

[59] K. Zhang, L. Wang, A. Pan, and B. B. Zhu. Smart Caching for Web Browsers. WWW '10.

FastRoute: A Scalable Load-Aware Anycast Routing Architecture for Modern CDNs

Ashley Flavel
Microsoft
ashleyfl@microsoft.com

Pradeepkumar Mani
Microsoft
prmani@microsoft.com

David A. Maltz
Microsoft
dmaltz@microsoft.com

Nick Holt
Microsoft
nickholt@microsoft.com

Jie Liu
Microsoft Research
jie.liu@microsoft.com

Yingying Chen
Microsoft
yinchen@microsoft.com

Oleg Surmachev
Microsoft
olegsu@microsoft.com

Abstract

Performance of online applications directly impacts user satisfaction. A major component of the user-perceived performance of the application is the time spent in transit between the user's device and the application existing in data centers. Content Delivery Networks (CDNs) are typically used to improve user-perceived application performance through a combination of caching and intelligent routing via proxies. In this paper, we describe *FastRoute*, a highly scalable and operational anycast-based system that has significantly improved the performance of numerous popular online services. While anycast is a common technique in modern CDNs for providing high-performance proximity routing, it sacrifices control over the load arriving at any individual proxy. We demonstrate that by collocating DNS and proxy services in each FastRoute node location, we can create a high-performance, completely distributed system for routing users to a nearby proxy while still enabling the graceful avoidance of overload an any individual proxy.

1 Introduction

The latency between user requests and computer reactions directly relates to user engagement and loyalty [11, 31]. With the expansion of applications connecting to cloud servers, the network latency between users and cloud servers becomes a major component of the overall application performance.

As network delays are largely proportional to the routing distance between clients and servers, application operators often employ services of Content Distribution Networks (CDNs). A CDN deploys proxy servers in geographically dispersed locations and then tunnels user requests through them. By utilizing these proxies, the CDN provides (among other things) performance improvements via techniques such as caching and split-TCP connections [27, 19].

While the overarching goal of latency reduction is universal, the logic to determine the proxy an individual user is routed to can be CDN-specific, based on what they offer to their customers, what the requirements of the application are, and what capacity limits the CDN has. Concentrating solely on the "optimal" proxy selection for every user based on latency can introduce additional complexity in the routing logic. In contrast, our design goals were two-fold, a) deliver a low-latency routing scheme that performed better than our existing CDN, and b) build an easy-to-operate, scalable and simple system

The operational aspect of the design goal results in an architecture that is willing to sacrifice some user performance in scenarios that occur rarely in order to maintain a simple design pattern. A major early design choice was to utilize anycast routing (see Section 2.2) as it enabled each FastRoute node (see Section 3.1) to operate independently of other FastRoute nodes (i.e. no real-time communication between nodes). Anycast routing has also successfully been used to deliver content by other CDNs including Edgecast and Cloudflare [28].

Although anycast routing has its advantages, there is the potential for an individual FastRoute node to become overloaded with user traffic (as the CDN does not control which proxy receives traffic, leaving it at the mercy of the intermixed routing policies of Internet Service Providers). To account for this, the FastRoute architecture utilizes multiple "layers" of FastRoute nodes — each with its own anycast IP (see Section 3.2.2). When a FastRoute node in a layer is overloaded, traffic is redirected to node(s) in the next layer. This layer-based approach is an example of choosing simplicity over user performance (for the expected, but rare overloaded node scenario[1]). I.e. instead of attempting to direct users from an overloaded node to a nearby node regardless of layer, we route users to the closest node in the next layer.

An artifact of using anycast routing is that the DNS

[1] If an overloaded node is not rare it indicates a build-out capacity issue.

must choose which anycast IP address to return to a client without knowing which proxy (of the ones announcing that address) the client's traffic will reach. Consequently, some intelligence is needed to determine which DNS responses must be redirected to the next layer. In Section 3.2.1 we show that by collocating DNS servers with proxy servers in the same node locations, a large percentage of user and DNS traffic land at the same a FastRoute node. Although the likelihood of a DNS query and its associated subsequent HTTP requests landing on the same node is not 100% (73% in our network), for the purposes of offloading traffic this has proven sufficient in production for over 2 years. Consequently, a FastRoute node only needs to know about its own load — preserving the independence of nodes that anycast provides.

Although load management is a critical component of the overall system, it is expected to operate rarely. In most situations, users will be routed to the first layer and their performance is based on the intermixed decisions of all ISPs in the Internet. Although technically possible to dynamically influence routing decisions in real-time (e.g. [8]), the system needed to do this would require significant development effort and most critically — introduce additional complexity. Consequently, in Section 4 we introduce several monitoring tools we use for analysing user performance offline that influence our peering policies and who we choose to peer with.

FastRoute has been in operation for several years improving the performance of a number of popular Internet applications. The initial move to FastRoute from a third-party CDN demonstrated noticeable performance improvements. Further, the simple load management implementation has handled all such overload scenarios since its inception, and its fully distributed nature has enabled us to quickly add new proxy locations – further improving our user performance.

Our contributions in this paper are the following novel and interesting aspects of FastRoute:

Architecture:

- A scalable architecture that is robust, simple to deploy and operate, and just complex enough to handle challenges such as overloaded proxies due to anycast routing
- A simple, yet unique DNS-based load management technique that leverages the self-correlation between proxy traffic and DNS traffic in a collocated DNS and proxy system
- Use of multiple anycast rings of FastRoute nodes for load absorption to prevent ping-ponging of load between overloaded proxies

Longitudinal Experimental Results from Production System:

- Data showing that FastRoute works effectively, even in the face of traffic that would overload a simpler system.
- Data showing that the DNS-to-Proxy correlation is relatively stable and high across time.
- Data showing performance (latency) improvements at the 75th and 95th percentiles with our initial limited set of FastRoute nodes, for customers of 10 ISPs in the USA.
- Data showing Anycast stability is sufficient to run a production system

2 Background

Content Distribution Networks direct users to their nearby proxies to improve their performance. In this section we will first examine two fundamental technologies that are core to many of the techniques used to route traffic to the optimal proxy. We will then examine several known techniques that CDNs utilize to help select and route users to a nearby proxy.

2.1 DNS

The Domain Name System (DNS) [23] translates human friendly names (such as `www.contoso.com`) into IP addresses (such as 1.2.3.4). DNS utilizes a hierarchical system where end-users consult recursive DNS resolvers that identify and query an authoritative DNS resolver to discover the translated IP address. The recursive DNS resolver caches the translation for the duration of the time-to-live (TTL) associated with the response. Other clients that utilize the same recursive DNS resolver will receive the same response for the duration of the TTL.

2.2 Anycast Routing

Anycast is a routing technique historically popular with DNS systems due to its inherent ability to spread DDOS traffic among multiple sites as well as provide low latency lookups. It utilizes the fact that routers running the de-facto standard inter-domain routing protocol in the Internet (BGP [30]), select the shortest (based on policy and the BGP decision process) of multiple routes to reach a destination IP prefix. Consequently, if multiple destinations claim to be a single destination, routers independently examine the characteristics of the multiple available routes and select the shortest one (according to the BGP path selection process). The effect of this is that individual users are routed to the closest location claiming to be the IP prefix (see [28] for a good explanation of Anycast routing). Note that latency is not a consideration in the BGP path selection process. However, by "tuning" anycast routing announcements and negotiating policies of peering ISPs (as in Section 4), BGP routing decisions can align with latency based routing.

2.3 Proxy Selection Techniques

FastRoute uses Anycast TCP to route a user to a proxy. In this section, we describe the general approach a CDN would use with Anycast TCP as well as examine several other alternatives (a summary is included in [9]).

2.3.1 Anycast TCP

Anycast TCP is used by many modern CDNs including Edgecast and CloudFlare[28]. This approach has all proxies responding on the same IP address and Internet routing protocols determine the closest proxy[8]. If a location cannot serve traffic, it withdraws its route and the Internet's routing protocols route users to the next closest location.

A difficulty with this approach is that control over where user traffic lands is relinquished to Internet routing protocols. Consequently, avoiding the overload of an individual proxy by controlling routes becomes challenging to accomplish in an automated fashion, as this either requires a traffic engineering technique such as [17, 29, 8] or a DNS based approach as presented in this paper.

A second concern with this approach is that a route change in the middle of a TCP session can result in the user communicating with an alternative proxy mid-session. Attempts can be made to direct rogue TCP flows and route them to the correct proxy[8]; however, much like [22], we analysed the availability of end-users (see Section 5.3), finding the availability of an anycast destination for a small file download was equivalent to unicast indicating this issue did not warrent implementing such a solution at the time of creation.

2.3.2 Anycast DNS

Utilizing an anycast based DNS has become the standard mechanism used by major providers to offer a high level of performance and denial-of-service protection. However, one mechanism a CDN can use to select which proxy to route traffic to takes advantage of information obtained from **where** the DNS lookup occurs [15]. By co-locating the DNS servers with each proxy, a request landing on a proxy simply returns the unicast IP of the collocated proxy. This is a simple solution that utilizes the Internet's routing protocols to find the shortest route to proxy location. However, with this approach, the 'closest' proxy selected is based on 'closeness' to the recursive DNS of the user instead of the 'closeness' to the user themselves. In practice we found the closest proxy for the DNS and user (self-correlation) for our network was the same for 73% of requests based on the analysis described in Section 3.2.1. Although we do not use this unicast based approach (due to it sacrificing the performance benefits of Anycast TCP), our architecture can easily be modified to return unicast IPs instead of anycast IPs (making the self correlation 100%).

2.3.3 Internet Map

An Internet map based approach relies on projecting the entire IP address space onto the set of available proxies (e.g. [13]). By collecting performance data either passively [20, 25], actively through probing mechanisms [12, 24], or statically mapping using geo-to-IP mappings [4], a map can be created that maps IP address space to proxy locations. The map is updated over time as network conditions change or proxies come up/down.

We did not pursue this approach (in contrast to the Anycast TCP approach) as it required global knowledge to analyse user latency data as well as real-time load and health metrics of nodes to decide where to route traffic. Further, the lack of granularity of DNS based responses [26] (which is the predominant method to route users based on an Internet map), the lack of motivation for ISPs (who still perform a majority of the DNS resolutions for users — 90% in the USA from our calculations) to implement more granular DNS requests [14, 18, 26] and the additional complexity introduced when supporting IPv6[2] when supporting IPv6 made this approach less appealing.

2.3.4 Other techniques

Several other techniques including manifest modification for video providers [7] (not applicable to other content types) or HTTP redirection are possible. The content we are delivering was predominantly dynamic content making the cost of a HTTP redirection high compared to transfer time making this approach infeasible.

3 FastRoute Architecture

In this section we describe a) the components within an individual FastRoute node b) why we can make local

[2] DNS requests are **not** guaranteed to be made over the same protocol for the answer they are requesting. I.e. an IPv4 (A record) resolution can be made over IPv6 and vice-versa. Consequently, supporting IPv6 introduces a 4x complexity over Ipv4 only by adding an additional 3 maps.

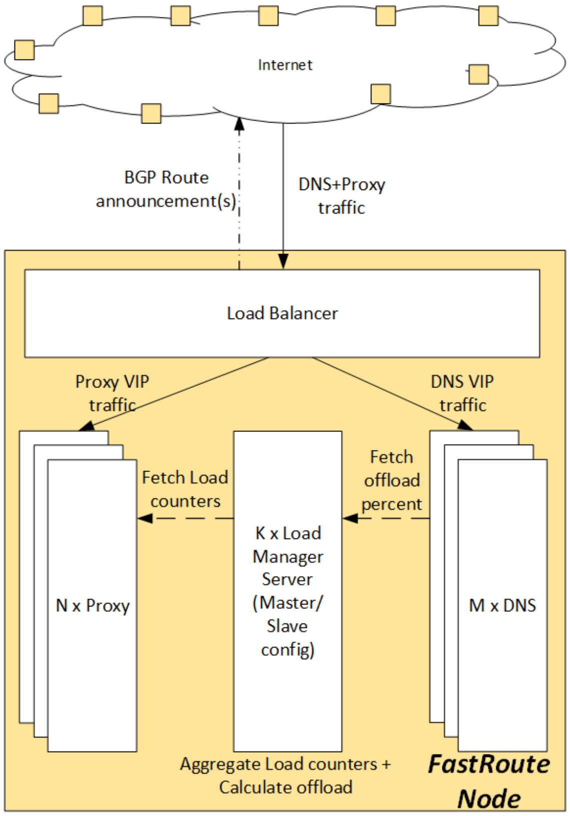


Figure 1: The FastRoute node architecture consists of 4 major components: Load Balancer (to balance traffic to a single Virtual IP (VIP) to multiple instances), DNS (to answer user DNS queries), Proxy (to serve application traffic) and Load Manager (to determine the offload percentage).

decisions on redirecting load away from a node and c) how the local algorithm makes its decisions.

3.1 FastRoute Node

In this section we describe the services within an individual FastRoute node and the communication between them. As explained in Section 3.2, no communication is needed outside an individual node to route users to a proxy.

In Figure 1 we show the four major stand-alone services that exist within a FastRoute node:- Load Balancer, Proxy, DNS and Load Manager. Each service may be on independent or co-existing on the same physical machines.

The Load Balancer is responsible for spreading user traffic between N instances of the Proxy and DNS traffic between M instances of the DNS service. When the number of healthy proxy or DNS services drops below a threshold, the anycast BGP prefixes of the DNS and proxy are withdrawn. Equivalently, when the number of healthy proxy and DNS services is higher than a threshold the BGP prefixes are announced. Announcing and withdrawing routes is the mechanism by which a FastRoute node either chooses to receive traffic or not.

The Proxy service is responsible for handling user traffic (e.g. terminating user TCP sessions, caching, fetching content from origin servers, blocking DDOS traffic etc). For each type of traffic it is handling, a counter defining the load is published locally.

The DNS service responds to each DNS query with one of two possible responses: either the anycast IP of its own FastRoute node or a CNAME (redirection) to the next layer (details included in Section 3.2). The probability of returning the CNAME is determined by reading at regular intervals a configuration published by the load management service.

The load management service is responsible for aggregating the counters collected across all proxy nodes and publishing the probability of returning the redirection CNAME for each DNS name. It operates in a master/slave configuration so all DNS services within the node receive the same offload probability. Details of the algorithm the load management service uses are included in Section 3.2.

3.2 Distributed Load Management

When no individual proxy is receiving more traffic than it is configured to handle, the operation of the system follows the pure anycast mechanism with all DNS requests being responded with the same IP address and the Internet's routing protocols determine the closest proxy for each user.

However, as described in Section 2.3.1, Internet routing protocols have no knowledge of the load on any individual proxy. Consequently, an individual proxy can be overloaded when using anycast if proxy locations aren't significantly over provisioned relative to the expected traffic load. The ability to over-provision such proxy locations is often limited as power and space comes at a premium. Further, the ability to add new capacity to overloaded locations has significant lead-time (can be weeks to months), hence we must have the ability to dynamically shift load in real-time - even if we expect this situation to be rare.

We considered two main techniques for Load Management - (1) altering BGP routes, and (2) modifying DNS responses. When an individual proxy is overloaded, utilizing BGP techniques such as AS Path pre-pending or withdrawing routes to one or more peers is one way to reduce the traffic on the proxy. However, such techniques are difficult to perfect as they can suffer from cascading failures (as an action from one proxy can cause a traffic swarm to a nearby proxy causing it to take action, and so on). A centralized system could be used to manage such actions; however, a significant amount of complexity would need to be introduced into predicting where traffic would route to, given a particular action. Further, taking BGP-level actions when utilizing an anycast based system results in route-churn and subsequently, an increased rate of TCP session breakages.

Conversely, modifying DNS responses enables the BGP topology to remain unchanged. This has two main attractive features. First, the BGP topology improvements and monitoring described in Section 4.1 remain independent to load management. Second, modifying DNS responses will only affect the routing of new users (as users already connected to a proxy will continue their session). Hence, DNS is a less abrupt change to users and a more gradual shift of overall traffic patterns than modifying BGP. However, there are two primary difficulties when using DNS for load management: first, the DNS server must infer that its response will result in additional traffic landing on an overloaded proxy (or not); second, given that a DNS server knows which DNS responses to modify to prevent load from landing on an overloaded proxy, what answer does it then respond with to redirect users causing the excessive load?

To solve the first difficulty, we used an artifact used by many traditional CDNs — the LDNS and the users of that LDNS are often in a similar network location. Consequently, we collocated our authoritative DNS servers and proxy servers in a FastRoute Node. Our hypothesis was that there would be a high correlation between the location of the proxy receiving the user traffic and the authoritative DNS server receiving the DNS request. Given a high correlation, by altering only the decision of the collocated DNS server, we can divert traffic and avoid overloading the proxy. This has a very appealing characteristic that the only communication needed is between the collocated proxy and DNS in a given FastRoute node. We discuss more on this communication in Section 3.3.2.

The second difficulty we encounter is where to direct traffic that would normally be directed to an overloaded proxy. One option is to return the unicast IP address of the next closest non-overloaded proxy, however, this in essence means creating a parallel system like the Internet map as described in Section 2.3.3 for the (expected) rare situation of an overloaded proxy. In contrast, we translate the problem from one of “where to send the traffic”, to one of “where not to send the traffic”. As we only have collocated proxy-DNS pair communication, the only load a DNS server is aware of is its own. Consequently, each DNS simply knows to direct traffic to “not-me”. By configuring multiple anycast IP addresses on different sets of proxies, the DNS server can direct traffic to one of the anycast IPs that it is not responsible for. However, when using such an approach, it is possible that multiple proxies experience high load and direct traffic to each other — causing more load on proxies that are already overloaded. The underlying problem is there is a possibility for ping-ponging of traffic among overloaded proxies, if we are not careful. We address this concern in Section 3.2.2, by setting up loop-free traffic diversion paths.

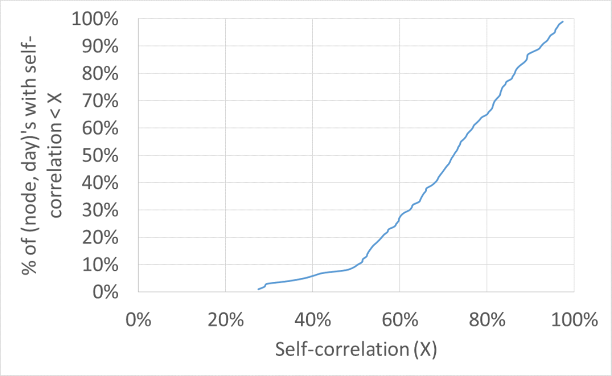


Figure 2: CDF of self-correlation values observed each day for all FastRoute nodes over a week. Each datapoint is a self-correlation for an individual node for a single day. 90% of datapoints have a self-correlation greater than 50% with no datapoint less than 27% justifying FastRoute’s decisoin to use onl local decisions for load management.

3.2.1 Local Controllability of Load

If DNS queries and subsequent user traffic to proxies lands on the same FastRoute Node, we call such user traffic as *controllable*. We measure correlation between two FastRoute nodes *i* and *j* as the likelihood of the DNS query landing on FastRoute node *i* (DNS response is the anycast IP of the proxy) and the subsequent user traffic landing on the proxy in node *j*. The *self-correlation* of any node *i* is a measure of the controllable load on that node. Every node could have a mix of controllable and uncontrollable load. From the data gathered using the approach shown in [16], we can construct a correlation matrix for all the nodes in the system. The diagonal of the correlation matrix gives the self-correlation values for the various nodes.

For our solution to be able to handle a given load, we rely on the self-correlation being high enough to offload sufficient traffic to avoid congestion. In Figure 2 we show the CDF of self-correlation values observed each day for every FastRoute node using over 10 million DNS-to-HTTP request mappings collected over a week in February 2015 from a representative sample of users of a major application utilizing FastRoute. Each node contributed around 7 data points (one self-correlation value per day) towards computing the CDF. Any FastRoute node receiving less than 100 samples on an individual day was excluded from the results.

We see that more than 90% of $(Node, day)$ pairs have a self-correlation greater than 50%. No node on any day had a self-correlation below 27%. Further, when examining the individual node self-correlation values, they remain relatively constant over the entire week.

For the nodes with self-correlation below 50% we found that the *cross-correlation* with either one or several other neighboring nodes was relatively high. For example, one node in Europe with a self-correlation of approximately 28% had four other nodes in nearby cities with cross-correlation values of $\tilde{2}0\%$, 18%, 17% and 10%. A distinct North American node also with a self-correlation of approximately 28% had a single other node with a cross-correlation value of 50%. We see this pattern (of a small subset of nodes that have high cross-correlation values) consistently throughout other nodes with relatively low self-correlation.

FastRoute does not currently attempt to do anything special for nodes with low self-correlation. This is based on our design principle of simplicity — do not build unnecessary complexity unless absolutely needed (and so far it has not). The two FastRoute nodes discussed above serve less than 2% of total global traffic and are sufficiently over-provisioned to handle the load they receive. However, if any node (low self-correlation or not) is unable to offload sufficient traffic, we have the ability to alert an operator to manually divert traffic from other nodes (based on the historic non-diagonal terms of the correlation matrix).

In the future, if operators are being involved sufficiently often enough to justify the additional complexity, we can implement one or more of the following features:-

- Lowly correlated nodes can "commit suicide" (i.e., withdraw DNS and Proxy anycast BGP routes) when an offered load is unable to be sufficiently diverted, resulting in traffic (expected) to land on nearby nodes with higher self-correlation values and can divert traffic if necessary. This keeps our current desirable system property of no real-time communication between nodes.

- Lowly correlated nodes can inform the small set of nearby nodes that have a high cross-correlation to start offloading traffic (e.g. in the examples presented earlier, this would increase the ability to offload traffic to 93% for the European node and 78% for the North American node). This breaks our current system property of no real-time communication between nodes, but does limit it to a small subset of nodes.

- Nodes with low self-correlation can be configured in "anycast-DNS" mode (i.e. DNS served over anycast, but proxy over unicast addresses; see Section 2.3.2). Such nodes could always be configured in this mode, or nodes could automatically transition to this mode when they cannot divert sufficient traffic.

- The proxy can take steps to divert traffic including reducing its workload (e.g. dropping lower priority traffic) or diverting traffic via HTTP 302 redirects.

As more FastRoute nodes are added, we will continue to monitor the correlation matrix to ensure it is sufficient to handle our traffic patterns.

3.2.2 Loop-free Diversion of Load

So far we have discussed the control over load landing on a proxy with purely local actions (The DNS altering its decision to divert traffic away from the collocated proxy). We now discuss how we determine what the altered response should be.

Our approach is one that utilizes anycast *layers* where each layer has a different anycast IP address for the DNS and proxy services. Each DNS knows only the domain name of its parent layer. Under load, it will start CNAME'ing requests to its parent layer domain name[3]. By utilizing a CNAME, we force the recursive resolver to fetch the DNS name resolution from a FastRoute node within the parent layer. This mechanism ensures that a parent layer node has control over traffic landing in the parent layer with the parent layer following the same process if it becomes overloaded.

We see an example setup of anycast layers in Figure 3. Here we see FastRoute nodes 1 and 2 in the outermost layer becoming overloaded. This results in both nodes diverting traffic to the middle layer resulting in additional traffic landing on nodes 3 and 4. Node 4 determines that it is now being overloaded as a result and diverts load to the innermost layer with node 5 receiving the additional traffic. From a user perspective, although their DNS requests may be bounced off several nodes, their proxy traffic will not experience the redirects.

Higher level layers are not required to be as close to users as lower level layers, consequently, they can be in physical locations where space is relatively cheap and easy to add capacity (e.g. within large data centers with elastic capacity [21, 1]). Hence, bursts of traffic can be

[3] A CNAME is an alias within the DNS protocol that causes the recursive resolver to undertake a new lookup

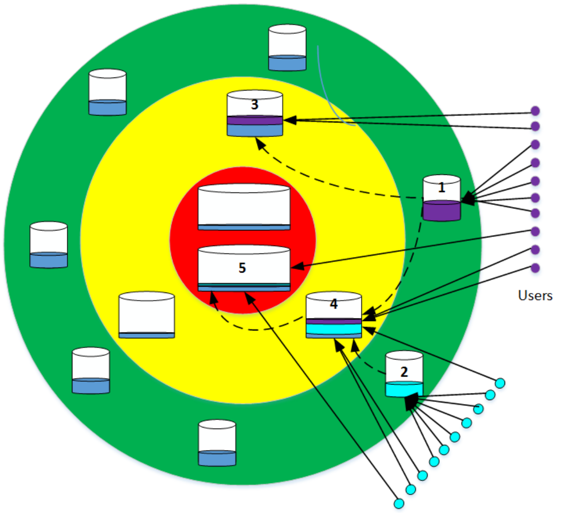


Figure 3: An example configuration with three Anycast *layers*. Solid arrows denote user connections, while dotted arrows denote the effect of diverting traffic by nodes that would otherwise be in overload.

handled by over-provisioning. By diverting lower priority traffic from higher layers first (as in Section 3.3.1) we can avoid the perceived user performance impact.

Although we have shown a single directed path between the lowest layer and highest layer, more advanced configurations are possible. Several extensions include an individual proxy may have two parent layers and offload proportionally between the layers (we did operate in this mode initially when the highest layer did not have sufficient spare capacity), different applications may have a different relationship between layers or individual proxies may exist in multiple layers (i.e. a layer may consist of locations that are subset of a lower layer). The only requirements are that the relationship between layers be loop-free and the highest layer be able to handle the load with no ability to divert traffic.

3.3 Local Offload Algorithm

In this section, we will discuss our approach to use DNS to manage load on a collocated proxy. We will begin by defining the notion of *load* first: user traffic hitting a proxy will consume various system resources such as CPU, memory, network bandwidth, etc., in the proxy. We refer to the strain placed on these resources as *load*. The nature of "load" could vary based on the traffic hitting each end point in the proxy (e.g. short HTTP request-response type queries are generally bottlenecked by CPU; file streaming applications are generally bottlenecked by network bandwidth, etc.). We can control "load" on a particular resource by controlling the user traffic hitting the end point(s) associated with the "load". For every such identified loaded resource associated with the end point (one resource per end point), we define an *overload threshold*, that defines the operating boundary of the proxy, and we consider the proxy overloaded if the load on any resource exceeds the threshold. The goal of FastRoute's load management scheme is to operate the system such that the load on any resource in a given proxy stays under the overload threshold. Also, as each FastRoute load manager instance expects a fraction of traffic that is not controllable locally, multiple instances of the load management service can operate on different endpoints hosted on the same physical machine — even if they utilize each other's bottlenecked resource (e.g. a filestreaming application may be bottlenecked by network bandwidth, but still consumes CPU). This behavior simply alters the fraction of uncontrollable load each load manager instance sees.

3.3.1 When to Divert Load?

In our design it is up to an individual node to discover when it is overloaded and divert some traffic. The load management algorithm that controls offload should be able to quickly recognize an overload situation and divert (just enough) load to another layer, so as to bring load under the overload threshold; equally important for the algorithm is to recognize that the overload situation has passed, and reduce or stop the offloading, as appropriate. Also, it is important to note that any delay in offloading during overload will cause measurable user impact (may cause service unavailability), while any delay in bringing back the traffic once overload has passed, has a relatively smaller penalty and user impact (e.g. higher latency due to being served from a farther layer). The two types of load to expect are:-

- Slowly increasing/decreasing load. This load is caused by the natural user patterns throughout the day. Generally, over a day a diurnal pattern is seen based on users' activities in the timezone of the proxy's catchment zone. Figure 4 shows diurnal traffic pattern observed over a period of 3 days in a proxy in production.
- Step changes in load. This is caused by a nearby proxy going down and all traffic from that proxy hitting the next closest proxy. We show an example of one such occurrence from our production system in Figure 5.

Consequently, our algorithm that determines which DNS answer to return must handle the above two scenarios. Challenges in this algorithm surround the limitations of the control mechanism. These include:-

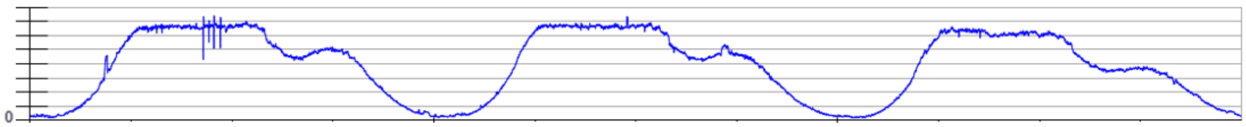


Figure 4: Traffic pattern over a period of 3 days for a single node. The Y-axis represents traffic volume. We have removed the values for confidentiality purposes

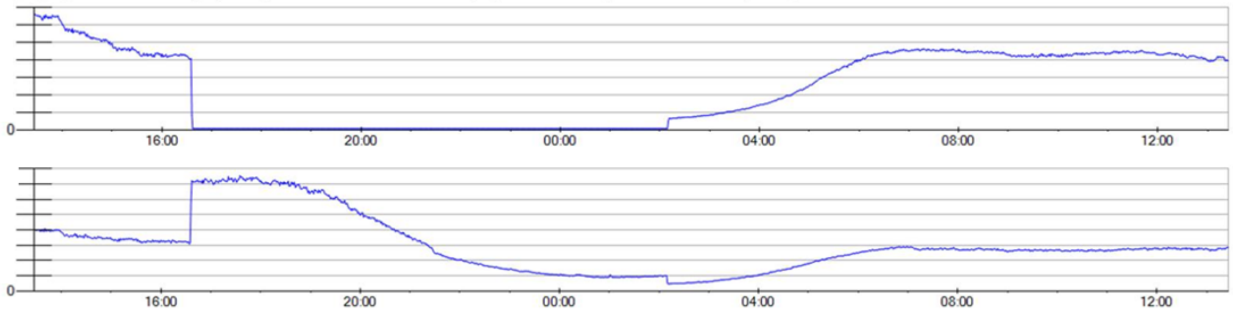


Figure 5: At around 17:00, a neighboring proxy (top) fails and as a result the closest proxy (bottom) is hit with all the load. The Y-axis represents traffic volume. We have removed the values for confidentiality purposes

- The TTL on a DNS response causes a delayed response to changes. Though it was shown in [16] that load management using DNS is feasible, the delay due to TTL is unavoidable.
- Local DNS servers have differing numbers of users behind them.
- A user's DNS lookup may not land on the same proxy as their TCP traffic (see Section 3.2.1 for analysis). Consequently, some load on a proxy (from its perspective) is uncontrollable.

Given the limitations of the control mechanism we have, we would like our control algorithm to be able to

- Quickly reduce load when a step change forces traffic above a desired utilization.
- Prioritize low value traffic to be offloaded
- Alert if the "uncontrollable" load becomes too large to maintain load under the desired utilization.

Many algorithms can support these characteristics. We present a simplified version of our algorithm in production. Let S be the current load on a given resource at node i, T be the overload threshold that is set as the operating boundary for this resource, under which we expect the proxy to operate at all times, and x be the fraction of traffic being offloaded to the next higher layer (offload probability). In order for the load management control loop to function effectively, the load sampling interval is set to higher than twice the TTL of the DNS responses (note that the TTL on the responses reflected the desire of responsiveness; i.e. longer TTL implied that a sustained overload condition and slower reaction to overload was acceptable).

- if $S > T$, the node i is overloaded. Offload probability x is increased super-linearly (maximum value = 1)
- if $S < T$, the node i is NOT overloaded. Offload probability x is decreased linearly (minimum value = 0)

As an extension, we implemented priority-based offloading of different end points that have the same load characteristics (no results shown in this paper). Among end points that contend for the same resource, we defined load management policies such that offload happens in some desired priority order. For example, say the proxy is the end point for both `http://www.foo.com` and `http://www.bar.com`, and traffic to either of these end points will use up CPU. Suppose, `http://www.foo.com` is more "important" than `http://www.bar.com`, and say the overload threshold is set to 70%. When overload occurs (i.e. CPU use exceeds 70%), the system will begin offloading customers of `http://www.bar.com` in an effort to control the load on the system. If the overload persists, then customers of `http://www.bar.com` are fully offloaded before offloading any customers of `http://www.foo.com`. If overload persists even after offloading 100% of customers of both endpoints, then manual intervention is sought by engaging personnel from the FastRoute operations team to artificially increase the measured load on highly cross-correlated neighboring nodes causing an increased diversion of traffic away from the overloaded node.

3.3.2 Scalability and Independent Operation

Given that (a) our operating assumption is that each node has sufficient self-correlation, and (b) the DNS and proxy are collocated in the FastRoute node, it thus follows that the load management system situated at any given node only needs to monitor the various aspects of load on the local node. Once it has collected the load data, the load management system computes the offload fraction for a given end point, and it only needs to communicate the results to the local DNS. Thus, all communication needed to make load management work effectively is contained fully within the same FastRoute node, without the need for any external input or sharing of global state, which makes the operation of FastRoute nodes completely independent of one another, and allows for simplified and easy-to-scale deployment.

4 Improving Anycast Routes over Time

We chose an anycast TCP based approach for FastRoute due to its simplicity and low dependence on DNS for optimal proxy selection. Consequently, we rely heavily on BGP (the de-facto standard inter-domain routing protocol used in the Internet) to best direct users to the closest proxy. The underlying assumption when using anycast is that the shortest route chosen by BGP is also the lowest latency route. However, due to the way the BGP route selection process operates, this may not always be the case. Although possible to implement a real-time system that adapts to the current network topology to modify route announcments of "flip-back" to unicast, this would introduce additional complexity — something we wished to avoid.

Consequently, we opt for a primarily offline approach to monitoring the behavior of anycast. We utilize the user performance telementry to analyse daily user performance (see Section 4.1) to prioritize network peering improvements and identify performance changes for a set of users (see Section 4.2). Availability is most critical, hence we monitor availability in real-time via active Internet based probes such as [3, 2, 5] and internal probes (from within the node itself).

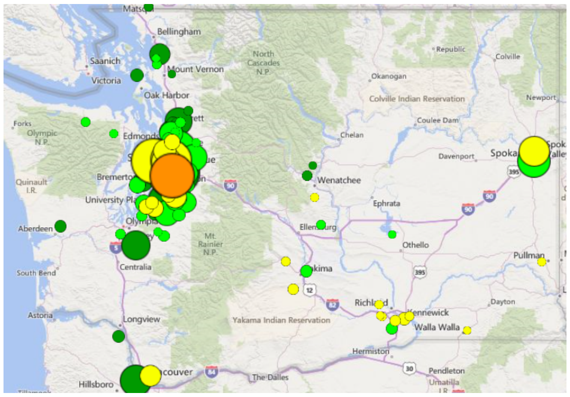


Figure 6: User performance grouped by ISP, geographic location and proxy location displayed on a Bing map. The size of the bubble represents the relative number of users. The color of the bubble represents the relative performance.

4.1 Identifying Performance Problems

One of the most valuable visualization techniques we developed as part of FastRoute was to overlay performance data collected from users of our production application(s) on top of a Bing map. Multiple views of user performance data were then plotted on top of this map providing unprecedented insight into how our users were experiencing our service. The most basic view we created using this technique is shown in Figure 6. Here we see users in Washington state. Navigation timing data [6] for these users are aggregated based on the user's geographic location, the ISP they are connected to and the proxy that they connect to. We display this data by sizing the bubble based on the relative number of users in the aggregate group and coloring the bubble based on the relative performance the users receive (red (worst), orange, yellow, light green, dark green (best)). From the example in Figure 6 we can quickly determine that we have a significant user base in the Seattle region (as expected due to the large population in this area)[4]. We can also see that one particular ISP is experiencing lower levels of performance than others in the same region. Upon further investigation (with data contained in the flyout box that appears when hovering over this bubble), we found this ISP was a cellular provider - expected to have slower performance than a cable or DSL network.

This display quickly identified large user populations that were receiving a lower level of performance than others. By filtering by individual proxies, it became immediately obvious when users were routed to a sub-optimal location (e.g. if European users were being routed to the North America). We found the performance of major ISPs to be relatively constant day-over-day. Consequently, by identifying the ISPs whose users

[4] Note that we have introduced random jitter around the actual geo-location of the user population to avoid bubbles being drawn directly on top of each other

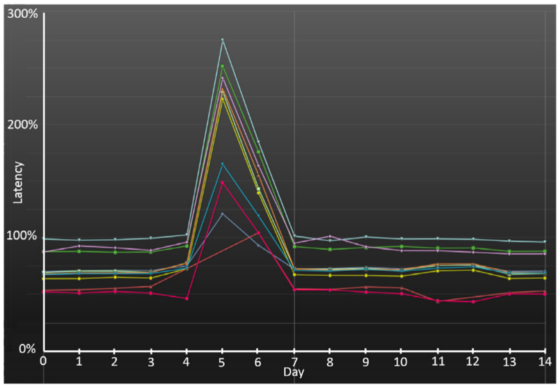


Figure 7: Latency vs Time for several ISPs. A peering link change on Day 5 resulted in a substantial increase in latency. Part-way through Day 6 the link was restored resulting in the expected performance returning.

were being sub-optimally routed (and prioritizing them based on user populations), our ISP peering team could prioritize their efforts to best satisfy our users (improving the performance of users accessing all applications of the Microsoft network - not just those running through FastRoute).

4.2 Identifying Changes in Performance

The above 'map' based view of performance is highly beneficial for analyzing a snapshot of performance. However, it is not as beneficial when trying to identify performance changes. Our goal is also to continually improve the performance for all our users. By considering the current performance of users as a benchmark, we can identify performance degradations, correlate the changes with known network changes and revert them if necessary. For example, in Figure 7 we see several ISPs' latency dramatically increase in the middle of the time series. This was as a result of an alteration in our peering relationship with another ISP resulting in congestion. By identifying an anomaly in the expected performance of users from this ISP, we were able to quickly rectify the issue, ensuring the effect on our users was minimized.

Conversely, the addition of new peering relationships and their impact on user performance was directly attributable providing business justification for the (possible) additional cost.

In a similar way, we can identify black-holing (or hijacking) of traffic (e.g. [10]). By monitoring the current user traffic volumes, we can identify anomalies in the expected volumes of traffic from particular ISPs.

4.3 Active Monitoring

Passive analysis of users reaching our service provide the best aggregate view of the performance our users are receiving. However, active probing mechanisms from third-party sources (e.g., [3, 2, 5]) provides additional sources of data. We found that utilizing systems that existed outside of our own infrastructure avoided circular dependencies and enables us to have information that is normally unavailable using passive monitoring (e.g. traceroutes).

5 FastRoute in Production

FastRoute was designed to replace a third-party CDN that was currently in operation for our Internet applications. However, in order to do so, we had to prove that FastRoute was not only functional, but there was a performance improvement and no drop in availability when compared to the third-party CDN. We describe in Section 5.1 how we compared the two systems, presenting data from our initial comparison.

A critical component of FastRoute is its ability to handle an overloaded proxy. This is expected to be a rare scenario given appropriate capacity planning, but prevents availability drops under load. In Section 5.2 we examine how load manager has operated in production.

A concern when using anycast is the availabilty of anycast in comparison to unicast given route flaps. In Section 5.3 we see no difference in the availability of a third-party unicast based CDN and our anycast solution.

5.1 Onboarding to FastRoute

We took a two step process for ensuring we reliably onboarded our first application onto FastRoute: first, compare availability and performance of non-production traffic served through FastRoute vs our existing third-party CDN, before gradually increasing the fraction of production traffic that was directed to FastRoute instead of the third-party CDN — ensuring real-user performance and availability was not degraded throughout the transition.

5.1.1 Non-Production Traffic Comparison

One method of comparison for two CDNs is through the use of active monitoring probes from agents spread throughout the Internet [2, 3, 5]. However, active probes come from a very limited set of locations and do not reflect the network location of our users. Consequently, we utilized our existing user base as our probing set. We achieved this by placing a small image on both the third party CDN as well as FastRoute. We then directed a small random sampling ($\sim 5\%$) of users to download the image from both the CDN and from FastRoute (after their page had loaded) and report the time taken (utilizing the javascript as described in [16]).

This demonstrated that FastRoute frequently delivered the image faster than our third-party CDN. This was sufficient justification to initiate the delivery of the actual application through FastRoute.

5.1.2 Production Traffic Comparison

The above non-production traffic experiment indicated that performance improvements were possible using FastRoute, however, there are many differences between a small image download and our production application. Consequently, we were cautious when moving to FastRoute. Our first production traffic moved onto FastRoute was a small percentage of a single US-based ISP. We configured our DNS servers to direct a random small fraction of users from the ISP's known LDNS IPs to FastRoute (leaving the remainder on the third-party CDN).

By analysing the performance data for the random sampling of users and comparing with the third-party CDN, we were able to ascertain the performance difference between the two CDNs[5]. This also enabled us to gather confidence that we were functionally equivalent to the third-party CDN. We repeated this "flighting" of different sets of users at different times and for different durations. We see in Figure 8 that for 10 major ISPs contributing more than 60% of traffic within the United States, all experienced a performance improvement with FastRoute. This initial comparison was undertaken with our initial deployment of only 10 FastRoute nodes throughout the United States[6]. This data was sufficient to justify increasing the percentage of users directed to FastRoute until 100% of users now pass through FastRoute. Since the time of analysis, we have increased the number of FastRoute nodes, added new applications and and improved our network connectivity to ISPs to further improve user performance.

5.2 Load Management in Production

We designed FastRoute's load manager with a single configurable parameter for each application — the threshold that a metric must be kept under (see Section 3). This metric is collected periodically and Load Manager reacts based on the current and previous values of the metric. We see in Figure 9 the traffic patterns of one application running on FastRoute. This application has a particularly spiky metric that had a threshold set to 70%.

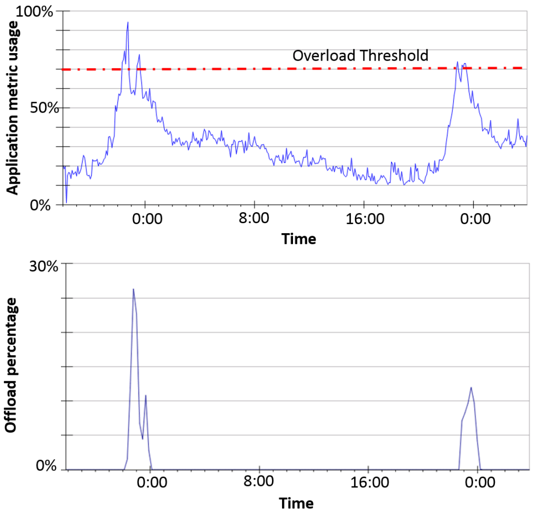


Figure 9: One application running on FastRoute had a very spiky traffic pattern within its diurnal. Load manager reacted automatically to divert the appropriate amount of traffic when load crossed the threshold, bringing it back when it had subsided sufficiently.

If the metric went above a hard limit of 100%, it would result in the loss of user traffic. We can see that the spiky nature of the burst in traffic resulted in the load manager offloading traffic quickly to bring the load back under the threshold. Some oscillation occurs around the threshold due to the delayed effects of DNS TTLs, but we control the traffic around the threshold

FastRoute's load management has been in operation for over 2 years. During this time we have seen a number of scenarios resulting in overloaded proxies (usually of the order of few incidents per week) including nearby proxies going down, naturally spiky user traffic patterns and code bugs in the proxy or DNS. FastRoute's load management scheme has provided the required safety net to handle all scenarios during this time without requiring manual intervention to modify routing policies or alter DNS configurations.

5.3 Anycast Availability

A concern when utilizing an anycast based solution is that the availability of the endpoint will be lower due to route fluctuations. In Figure 10 we show results from a synthetic test where approximately $20,000$ Bing toolbar clients downloaded a small image from an anycast IP announced from all 12 nodes (full set of nodes at time of experiment) throughout the Internet and the same image from a 3rd party (unicast) CDN over a period of a week. Although one datapoint showed the anycast availability

[5] Note that the performance improvement shown is for the entire FastRoute system (user to proxy to data center) not just for proxy selection.

[6] Nodes throughout the world were present, but for this analysis we focus on the United States.

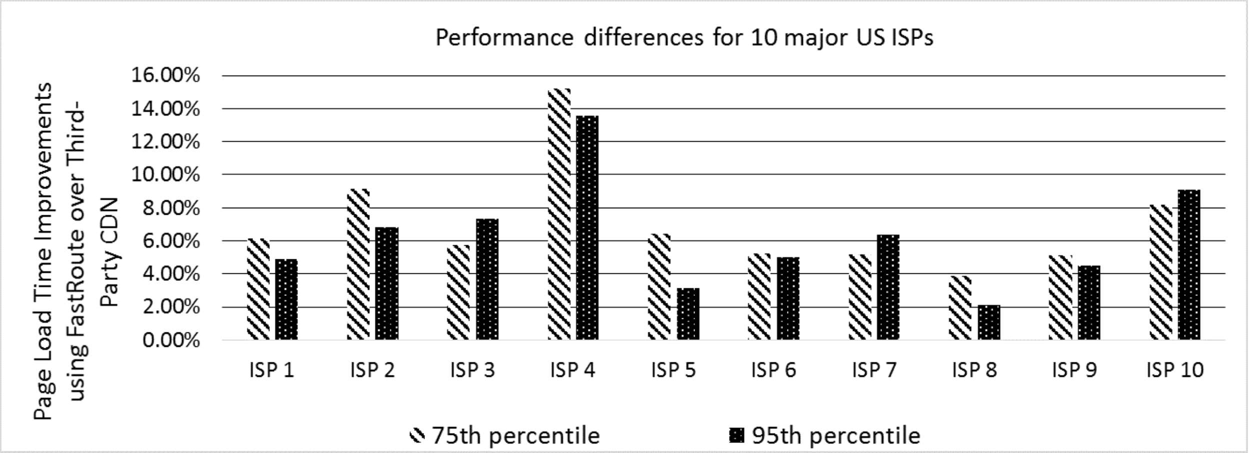


Figure 8: Performance improvements in 10 major US ISPs (contributing above 60% of all user traffic in the US) when using FastRoute compared to a third-party CDN. This data was collected when only 10 FastRoute nodes were in operation and no nodes were overloaded.

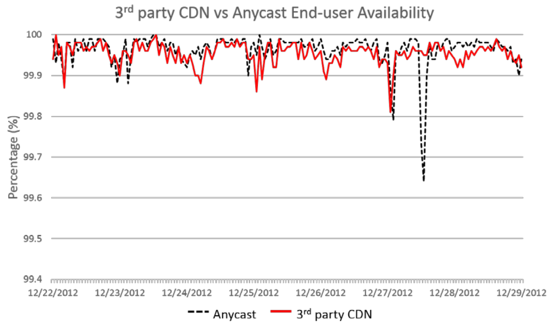


Figure 10: Anycast availability over a week compared to third-party CDN. Note the y-axis starts at 99.4%. The availabilty over the entire week was 99.96% vs 99.95% respectively.

dropped to 99.65% availabilty, we saw overall availabilities of 99.96% and 99.95% for anycast and third-party CDN availabilities respectively. These results, the success of other anycast TCP based CDNs (e.g. Cloudflare, Edgecast), the work presented in [22] and the lack of issues found in over 2 years serving production traffic (even as the set of nodes grows) indicate that anycast in the Internet is stable enough to run a production network on.

6 Future Work

Within this paper we have described the architecture of FastRoute, concentrating on proxy selection mechanism. Future work includes:

- Examining the selection of data center for user traffic landing on a proxy as well as techniques used to prioritize and multiplex user traffic to achieve optimal performance.
- Analyzing the impact to the self-correlation of DNS and proxy traffic when supporting IPv6.
- Analyzing the impact that local decisions made when diverting load, have on the global traffic patterns. In particular, we would like to understand the degree of sub-optimality introduced due to making local decisions, compared to making globally optimal decisions centrally.
- Studying the distributed load management algorithm from a control-theoretic perspective, and understand limits on correlation and user-traffic for stable system operation.

7 Conclusion

We have presented FastRoute, an anycast routing architecture for CDNs that is operational and provides high performance for users. FastRoute's architecture is robust, simple to deploy and operate, scalable, and just complex enough to handle overload conditions due to anycast routing. We highlighted performance gains obtained from our production system when routing users through FastRoute instead of a major third-party CDN.

We described a novel load management technique in FastRoute, which used the anycast DNS and multiple anycast proxy rings for load absorption. Excess traffic from one layer was directed to another higher layer using the collocated DNS. We provided data from our production system which showed that the correlation between DNS and corresponding user queries landing on the same node in our network to be sufficiently high, and relatively stable over time, both of which are crucial for effective load management. FastRoute load management has protected our production system numerous times from having an overload-induced outage, in addition to saving precious operator hours that would've otherwise been needed in a manual system. We provided one such example from our production system where proxy overload was dealt with quickly and effectively by FastRoute's load management.

Overall, FastRoute was designed with high performance, reliability and ease of operations in mind. By not over-complicating the design to handle rare scenarios — and trading off performance for simplicity to handle such rare scenarios — we were able to quickly adapt to new requirements with minimal development effort. We believe this is the biggest learning from the design, development, deployment and operation of FastRoute.

8 Acknowledgments

Many people were involved in the design of FastRoute. The authors would like to acknowledge the following people for their contribution to FastRoute: Dongluo Chen, Chao Zhang, Arun Navasivasakthivelsamy, Sergey Puchin, Bin Pang, Saurabh Mahajan, Darren Shakib, Kyle Peltonan, Mohit Suley, Ernie Chen, Aaron Heady, Brian Jack, Jeff Cohen, Randy Kern, Matthew Calder, Andrii Vasylevskyi, Narasimhan Venkataramaiah, Ashwin Sarin, Peter Oosterhof, Andy Lientz, Mark Kasten, Kevin Epperson, Christian Nielson, Dan Eckert.

We would also like to thank our shepherd Ethan Katz-Bassett.

References

[1] Amazon aws autoscaling. http://aws.amazon.com/autoscaling/.

[2] Gomez networks. www.gomeznetworks.com.

[3] Keynote. www.keynote.com.

[4] Neustar ip intelligence. http://www.neustar.biz/enterprise/ip-intelligence.

[5] Thousandeyes. www.thousandeyes.com.

[6] W3C navigation timing. http://www.w3.org/TR/navigation-timing/.

[7] V. K. Adhikari, Y. Guo, F. Hao, V. Hilt, and Z. Zhang. A tale of three cdns: An active measurement study of hulu and its cdns. *Global Internet Symposium*, 2012.

[8] H. Alzoubi, S. Lee, M. Rabinovich, O. Spatscheck, and J. V. der Merwe. A practical architecture for an anycast cdn. *ACM Transactions on the Web*, 2011.

[9] A. Barbir, B. Cain, R. Nair, and O. Spatscheck. RFC3568: Known content network request-routing mechanisms, July 2003.

[10] M. Brown. Renesys blog: Pakistan hijacks youtube, 2008. http://velocityconf.com/velocity2009/public/schedule/detail/8523.

[11] J. Brutlag. Speed matters. http://googleresearch.blogspot.com/2009/06/speed-matters.html.

[12] C. Bueno. Doppler: Internet radar, July 2011. https://www.facebook.com/notes/carlos-bueno/doppler-internet-radar/10150212498738920.

[13] M. Calder, X. Fan, Z. Hu, R. Govindan, J. Heidemann, and E. Katz-Bassett. Mapping the expansion of google's serving infrastructure. *IMC*, 2013.

[14] C. Contavalli, W. van der Gaast, S. Leach, and D. Rodden. Client subnet in dns requests. http://tools.ietf.org/html/draft-vandergaast-edns-client-subnet-02.

[15] A. Flavel, A. Karasaridis, and J. Miros. System and method for content delivery using dynamic region assignment. US Patent #20120290693, 2011.

[16] A. Flavel, P. Mani, and D. Maltz. Re-evaluating the responsiveness of dns-based network control. *IEEE LANMAN*, 2014.

[17] R. Gao, C. Dovrolis, and E. Zegura. Interdomain ingress traffic engineering through optimized as-path prepending. *Networking*, 2005.

[18] C. Huang, I. Batanov, and J. Li. A Practical Solution to the client-LDNS mismatch problem. *ACM SIGCOMM CCR*, 2012.

[19] R. U. Ibm and D. Rosu. An evaluation of tcp splice benefits in web proxy server. *WWW*, 2002.

[20] R. Krishnan, H. Madhyastha, S. Srinivasan, S. Jain, A. Krishnamurthy, T. Anderson, and J. Gao. Moving beyond end-to-end path information to optimize cdn performance. *IMC*, 2009.

[21] F. Lardinois. Microsoft adds auto scaling to windows azure, June 2013. http://techcrunch.com/2013/06/27/microsoft-adds-auto-scaling-to-windows-azure/.

[22] M. Levine, B. Lyon, and T. Underwood. TCP Anycast - Don't believe the FUD. *Proceedings of NANOG*, 2006.

[23] P. Mockapetris. Domain names - implementation and specification. *RFC1035*, 1987.

[24] R. Mukhtar and Z. Rosberg. A client side measurement scheme for request routing in virtual open content delivery networks. *Performance, Computing and Communications Conference*, 2003.

[25] E. Nygren, R. Sitaraman, and J. Sun. The akamai network: a platform for high performance internet applications. *ACM SIGOPS OSR*, 2010.

[26] J. Otto, M. Sanchez, J. Rula, and F. Bustamante. Content delivery and the natural evolution of dns: remote dns trends, performance issues and alternative solutions. *IMC*, 2012.

[27] A. Pathak, Y. A. Wang, C. Huang, A. Greenberg, Y. C. Hu, R. Kern, J. Li, and K. Ross. Measuring and evaluating tcp splitting for cloud services. *PAM*, 2009.

[28] M. Prince. A brief primer on anycast, 2011. `https://blog.cloudflare.com/a-brief-anycast-primer/`.

[29] B. Quoitin, S. Uhlig, C. Pelsser, C. Pelsser, L. Swinnen, and O. Bonaventure. Interdomain traffic engineering with bgp. *IEEE Communications Magazine*, 41, 2003.

[30] Y. Rekhter, T. Li, and S. Hares. RFC4271: A border gateway protocol 4, January 2006.

[31] E. Schurman and J. Brutlag. Performance related changes and their user impact. `http://velocityconf.com/velocity2009/public/schedule/detail/8523`.

PCC: Re-architecting Congestion Control for Consistent High Performance

Mo Dong[*], Qingxi Li[*], Doron Zarchy[**], P. Brighten Godfrey[*], and Michael Schapira[**]

[*]University of Illinois at Urbana-Champaign
[**]Hebrew University of Jerusalem

Abstract

TCP and its variants have suffered from surprisingly poor performance for decades. We argue the TCP family has little hope of achieving consistent high performance due to a fundamental architectural deficiency: hardwiring packet-level events to control responses. We propose Performance-oriented Congestion Control (PCC), a new congestion control architecture in which each sender continuously observes the connection between its *actions* and *empirically experienced performance*, enabling it to consistently adopt actions that result in high performance. We prove that PCC converges to a stable and fair equilibrium. Across many real-world and challenging environments, PCC shows consistent and often 10× performance improvement, with better fairness and stability than TCP. PCC requires no router hardware support or new packet format.

1 Introduction

In the roughly 25 years since its deployment, TCP's congestion control architecture has been notorious for degraded performance. TCP performs poorly on lossy links, penalizes high-RTT flows, underutilizes high bandwidth-delay product (BDP) connections, cannot handle rapidly changing networks, can collapse under data center incast [24] and incurs very high latency with bufferbloat [28] in the network.

As severe performance problems have accumulated over time, protocol "patches" have addressed problems in specific network conditions such as high BDP links [31,52], satellite links [23,42], data center [18,55], wireless and lossy links [38,39], and more. However, the fact that there are so many TCP variants suggests that each is only a point solution: they yield better performance under specific network conditions, but break in others. Worse, we found through real-world experiments that in many cases these TCP variants' performance is *still far away from optimal even in the network conditions for which they are specially engineered.* Indeed, TCP's low performance has impacted industry to the extent that there is a lucrative market for special-purpose high performance data transfer services [1,2,11,13].

Thus, the core problem remains largely unsolved: *achieving consistently high performance over complex real-world network conditions*. We argue this is indeed a very difficult task within TCP's rate control architecture, which we refer to as **hardwired mapping**: certain predefined packet-level events are hardwired to certain predefined control responses. TCP reacts to events that can be as simple as "one packet loss" (TCP New Reno) or can involve multiple signals like "one packet loss and RTT increased by $x\%$" (TCP Illinois). Similarly, the control response might be "halve the rate" (New Reno) or a more complex action like "reduce the window size w to $f(\Delta RTT)w$" (Illinois). The defining feature is that the control action is a direct function of packet-level events.

A hardwired mapping has to make *assumptions* about the network. Take a textbook event-control pair: a packet loss halves the congestion window. TCP *assumes* that the loss indicates congestion in the network. When the assumption is violated, halving the window size can severely degrade performance (e.g. if loss is random, rate should stay the same or increase). It is fundamentally hard to formulate an "always optimal" hardwired mapping in a complex real-world network because the actual optimal response to an event like a loss (i.e. decrease rate or increase? by how much?) is sensitive to network conditions. And modern networks have an immense diversity of conditions: random loss and zero loss, shallow queues and bufferbloat, RTTs of competing flows varying by more than 1000×, dynamics due to mobile wireless or path changes, links from Kbps to Gbps, AQMs, software routers, rate shaping at gateways, virtualization layers and middleboxes like firewalls, packet inspectors and load balancers. These factors add complexity far beyond what can be summarized by the relatively simplistic assumptions embedded in a hardwired mapping. Most unfortunately, when its assumptions are violated, TCP still rigidly carries out the harmful control action.

We propose a new congestion control architecture: Performance-oriented Congestion Control (PCC). PCC's goal is to understand what rate control actions improve performance based on *live experimental evidence*, avoiding TCP's assumptions about the network. PCC sends at a rate r for a short period of time, and observes the results (e.g. SACKs indicating delivery, loss, and latency of each packet). It aggregates these packet-level events into a utility function that describes an objective like "high throughput and low loss rate". The result is a single numerical performance utility u. At this point, PCC has run a single "micro-experiment" that showed sending at rate r produced utility u. To make a rate control decision, PCC runs multiple such micro-experiments: it tries sending at two different rates, and moves in the direction that empirically results in greater performance utility. This is effectively *A/B testing for rate control* and is the core of PCC's decisions. PCC runs these micro-experiments continuously (on every byte of data, not on occasional probes), driven by an online learning algorithm that tracks the empirically-optimal sending rate. Thus, rather than making assumptions about the potentially-complex network, PCC adopts the actions that *empirically* achieve consistent high performance.

PCC's rate control is selfish in nature, but surprisingly, using a widely applicable utility function, competing PCC senders provably converge to a fair equilibrium (with a single bottleneck link). Indeed, experiments show PCC achieves similar convergence time to TCP with significantly smaller rate variance. Moreover, the ability to express different objectives via choice of the utility function (e.g. throughput or latency) provides a flexibility beyond TCP's architecture.

With handling real-world complexity as a key goal, we experimentally evaluated a PCC implementation in large-scale and real-world networks. Without tweaking its control algorithm, PCC achieves consistent high performance and significantly beats *specially engineered* TCPs in various network environments: **(a.)** in the wild on the global commercial Internet (often more than **10×** the throughput of TCP CUBIC); **(b.)** inter-data center networks (**5.23×** vs. TCP Illinois); **(c.)** emulated satellite Internet links (**17×** vs TCP Hybla); **(d.)** unreliable lossy links (**10 − 37×** vs Illinois); **(e.)** unequal RTT of competing senders (an **architectural cure** to RTT unfairness); **(f.)** shallow buffered bottleneck links (up to **45×** higher performance, or **13×** less buffer to reach 90% throughput); **(g.)** rapidly changing networks (**14×** vs CUBIC, **5.6×** vs Illinois). PCC performs similar to ICTCP [55] in **(h.)** the incast scenario in data centers. Though it is a substantial shift in architecture, PCC can be deployed by only replacing the sender-side rate control of TCP. It can also deliver real data today with a user-space implementation at `speedier.net/pcc`.

2 PCC Architecture

2.1 The Key Idea

Suppose flow f is sending a stream of data at some rate and a packet is lost. How should f react? Should it slow the sending rate, or increase, and by how much? Or leave the rate unchanged? This is a difficult question to answer because real networks are complex: a single loss might be the result of *many* possible underlying network scenarios. To pick a few:

- f may be responsible for most of congestion. Then, it should decrease its rate.
- f might traverse a shallow buffer on a high-BDP link, with the loss due to bad luck in statistical multiplexing rather than high link utilization. Then, backing off a little is sufficient.
- There may be a higher-rate competing flow. Then, f should maintain its rate and let the other back off.
- There may be random non-congestion loss somewhere along the path. Then, f should maintain or increase its rate.

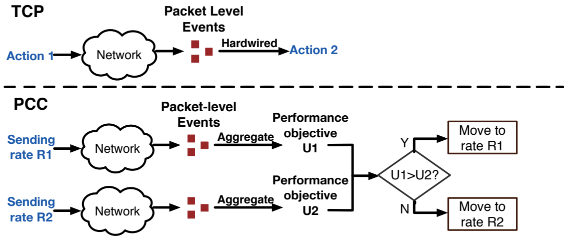


Figure 1: ***The decision-making structure of TCP and PCC.***

Classically, TCP assumes a packet loss indicates non-negligible congestion, and that halving its rate will improve network conditions. However, this assumption is false and will degrade performance in three of the four scenarios above. Fundamentally, picking an optimal *predefined and hardwired* control response is hard because for the same packet-level events, a control response optimal under one network scenario can decimate performance in even a slightly different scenario. The approach taken by a large number of TCP variants is to use more sophisticated packet-level events and control actions. But this does not solve the fundamental problem, because the approach *still hardwires predetermined events to predetermined control responses*, thus inevitably embedding unreliable assumptions about the network. When the unreliable assumptions are violated by the complexity of the network, performance degrades severely. For example, TCP Illinois [38] uses both loss and delay to form an event-control mapping, but its throughput collapses with even a small amount of random loss, or when network conditions are dynamic (§4). More examples are in §5.

Most unfortunately, if some control actions are indeed harming performance, TCP can still blindly "jump off the cliff", because it does not notice the control action's actual effect on performance.

But that observation points toward a solution. Can we design a control algorithm that directly understands whether or not its actions actually improve performance?

Conceptually, no matter how complex the network is, if a sender can directly measure that rate r_1 results in better performance than rate r_2, it has some evidence that r_1 is better than sending at r_2 — at least for this one sender. This example illustrates the key design rationale behind **Performance-oriented Congestion Control** (PCC): PCC makes control decisions based on *em-*

pirical evidence pairing ***actions*** *with directly* ***observed performance*** *results.*

PCC's control action is its choice of sending rate. PCC divides time into continuous time periods, called *monitor intervals* (MIs), whose length is normally one to two RTTs. In each MI, PCC tests an action: it picks a sending rate, say r, and sends data at rate r through the interval. After about an RTT, the sender will see selective ACKs (SACK) from the receiver, just like TCP. However, it does not trigger any predefined control response. Instead, PCC aggregates these SACKs into meaningful performance metrics including throughput, loss rate and latency. These performance metrics are combined to a numerical utility value, say u, via a *utility function*. The utility function can be customized for different data transmission objectives, but for now the reader can assume the objective of "high throughput and low loss rate", such as $u = T - L$ (where $T =$ throughput and $L =$ loss rate) which will capture the main insights of PCC. The end result is that PCC knows when it sent at rate r, it got utility of u.

The preceding describes a single "micro-experiment" through which PCC associates a specific *action* with an observed *resulting utility*. PCC runs these micro-experiments continuously, comparing the utility of different sending rates so it can track the optimal action over time. More specifically, PCC runs an online learning algorithm similar to gradient ascent. When starting at rate r, it tests rate $(1+\varepsilon)r$ and rate $(1-\varepsilon)r$, and moves in the direction (higher or lower rate) that empirically yields higher utility. It continues in this direction as long as utility continues increasing. If utility falls, it returns to a decision-making state where it again tests both higher and lower rates to find which produces higher utility.

Note that PCC does not send occasional probes or use throwaway data for measurements. It observes the results of its actual control decisions on the application's real data and does not pause sending to wait for results.

We now return to the example of the beginning of this section. Suppose PCC is testing rate 100 Mbps in a particular interval, and will test 105 Mbps in the following interval. If it encounters a packet loss in the first interval, will PCC increase or decrease? In fact, there is no specific event in a single interval that will always cause PCC to increase or decrease its rate. Instead, PCC will calculate the utility value for each of these two intervals, and move in the direction of higher utility. For example:

- If the network is congested as a result of this flow, then it is likely that sending at 100 Mbps will have similar throughput and lower loss rate, resulting in higher utility. PCC will decrease its rate.
- If the network is experiencing random loss, PCC is likely to find that the period with rate 105 Mbps has similar loss rate and slightly higher throughput, resulting in higher utility. PCC will therefore increase its rate despite the packet loss.

Throughout this process, PCC makes no assumptions about the underlying network conditions, instead observing which actions empirically produce higher utility and therefore achieving consistent high performance.

Decisions with noisy measurements. PCC's experiments on the live network will tend to move its rate in the direction that improves utility. But it may also make some incorrect decisions. In the example above, if the loss is random non-congestion loss, it may randomly occur that loss is substantially higher when PCC tests rate 105 Mbps, causing it to pick the lower rate. Alternately, if the loss is primarily due to congestion from this sender, unpredictable external events (perhaps another sender arriving with a large initial rate while PCC is testing rate 100 Mbps) might cause a particular 105 Mbps microexperiment to have higher throughput and lower loss rate. More generally, the network might be changing over time for reasons unrelated to the sender's action. This adds noise to the decision process: PCC will on average move in the right direction, but may make some unlucky errors.

We improve PCC's decisions with **multiple randomized controlled trials (RCTs)**. Rather than running two tests (one each at 100 and 105 Mbps), we conduct four in randomized order—e.g. perhaps $(100, 105, 105, 100)$. PCC only picks a particular rate as the winner if utility is higher in *both* trials with that rate. This produces increased confidence in a causal connection between PCC's action and the observed utility. If results are inconclusive, so each rate "wins" in one test, then PCC maintains its current rate, and we may have reached a local optimum (details follow later).

As we will see, without RCTs, PCC already offers a dramatic improvement in performance and stability compared with TCP, but RCTs further reduce rate variance by up to 35%. Although it might seem that RCTs will double convergence time, this is not the case because they help PCC make *better* decisions; overall, RCTs improve the stability/convergence-speed tradeoff space.

Many issues remain. We next delve into fairness, convergence, and choice of utility function; deployment; and flesh out the mechanism sketched above.

2.2 Fairness and Convergence

Each PCC sender optimizes its utility function value based only on locally observed performance metrics. However, this local selfishness does not imply loss of global stability, convergence and fairness. We next show that when selfish senders use a particular "safe" utility function and a simple control algorithm, they provably converge to fair rate equilibrium.

We assume n PCC senders $1, \ldots, n$ send traffic across a bottleneck link of capacity $C > 0$. Each sender i chooses its sending rate x_i to optimize its utility function

u_i. We choose a utility function expressing the common application-level goal of "high throughput and low loss":

$$u_i(x_i) = T_i \cdot Sigmoid_\alpha(L_i - 0.05) - x_i \cdot L_i$$

where x_i is sender i's sending rate, L_i is the observed data loss rate, $T_i = x_i(1 - L_i)$ is sender i's throughput, and $Sigmoid_\alpha(y) = \frac{1}{1+e^{\alpha y}}$ for some $\alpha > 0$ to be chosen later.

The above utility function is derived from a simpler starting point: $u_i(x_i) = T_i - x_i \cdot L_i$, i.e., i's throughput minus the production of its loss rate and sending rate. However, this utility function will make loss rate at equilibrium point approach 50% when the number of competing senders increases. Therefore, we include the sigmoid function as a "cut-off". When α is "big enough", $Sigmoid_\alpha(L_i - 0.05)$ will rapidly get closer to 0 as soon as L_i exceeds 0.05, leading to a negative utility for the sender. Thus, we are setting a barrier that caps the overall loss rate at about 5% in the worst case.

Theorem 1 *When $\alpha \geq \max\{2.2(n-1), 100\}$, there exists a unique stable state of sending rates $x_1^*, \ldots, x_n^*$ and, moreover, this state is fair, i.e., $x_1^* = x_2^* = \ldots = x_n^*$.*

To prove Theorem 1, we first prove that $\Sigma_j x_j$ will always be restricted to the region of $(C, \frac{20C}{19})$. Under this condition, our setting can be formulated as a concave game [46]. This enables us to use properties of such games to conclude that a unique rate equilibrium exists and is fair, i.e. $x_1^* = x_2^* = \ldots = x_n^*$. (Full proof: [6])

Next, we show that a simple control algorithm can converge to that equilibrium. At each time step t, each sender j updates its sending rate according to $x_j^{t+1} = x_j^t(1+\varepsilon)$ if j's utility would improve if it unilaterally made this change, and $x_j^{t+1} = x_j^t(1-\varepsilon)$ otherwise. Here $\varepsilon > 0$ is a small number ($\varepsilon = 0.01$, in the experiment). In this model, senders concurrently update their rates, but each sender decides based on a utility comparison as if it were the only one changing. This model does not explicitly consider measurement delay, but we believe it is a reasonable simplification (and experimental evidence bears out the conclusions). We also conjecture the model can be relaxed to allow for asynchrony. We discuss in §3 our implementation with practical optimizations of the control algorithm.

Theorem 2 *If all senders follow the above control algorithm, for every sender j, x_j converges to the domain $(\hat{x}(1-\varepsilon)^2, \hat{x}(1+\varepsilon)^2)$, where $\hat{x}$ denotes the sending rate in the unique stable state.*

It might seem surprising that PCC uses *multiplicative* rate increase and decrease, yet achieves convergence and fairness. If TCP used MIMD, in an idealized network senders would often get the same back-off signal at the same time, and so would take the *same multiplicative decisions in lockstep*, with the ratio of their rates never changing. In PCC, senders make *different* decisions. Consider a 100 Mbps link with sender A at rate 90 Mbps and B at 10 Mbps. When A experiments with slightly higher and lower rates $(1 \pm \varepsilon)90$ Mbps, it will find that it should decrease its rate to get higher utility because when it sends at higher than equilibrium rate, the loss rate dominates the utility function. However, when B experiments with $(1 \pm \varepsilon)10$ Mbps it finds that loss rate increase is negligible compared with its improved throughput. This occurs precisely because B is responsible for little of the congestion. In fact, this reasoning (and the formal proof of the game dynamics) is *independent of the step size* that the flows use in their experiments: PCC senders move towards the convergence point, even if they use a heterogeneous mix of AIMD, AIAD, MIMD, MIAD or other step functions. Convergence behavior does depend on the choice of utility function, however.

2.3 Utility Function: Source of Flexibility

PCC carries a level of flexibility beyond TCP's architecture: the same learning control algorithm can cater to different applications' heterogeneous objectives (e.g. latency vs. throughput) by using different utility functions. For example, under TCP's architecture, latency based protocols [38, 52] usually contain different hardwired mapping algorithms than loss-based protocols [31]. Therefore, without changing the control algorithm, as Sivaraman et al. [47] recently observed, TCP has to rely on different in-network active queue management (AQM) mechanisms to cater to different applications' objectives because *even with fair queueing*, TCP is blind to applications' objectives. However, by literally changing one line of code that describes the utility function, PCC can flip from "loss-based" (§2.2) to "latency-based" (§4.4) and thus caters to different applications' objectives without the complexity and cost of programmable AQMs [47]. That said, alternate utility functions are a largely unexplored area of PCC; in this work, we evaluate alternate utility functions only in the context of fair queueing (§4.4).

2.4 Deployment

Despite being a significant architectural shift, PCC needs only isolated changes. **No router support:** unlike ECN, XCP [35], and RCP [25], there are no new packet fields to be standardized and processed by routers. **No new protocol:** The packet format and semantics can simply remain as in TCP (SACK, hand-shaking and etc.). **No receiver change:** TCP SACK is enough feedback. What PCC does change is the control algorithm within the sender.

The remaining concern is how PCC safely replaces and interacts with TCP. We observe that there are many scenarios where critical applications suffer severely from TCP's poor performance and PCC can be safely deployed by fully replacing or being isolated from TCP.

First, **when a network resource is owned by a single entity** or can be reserved for it, the owner can replace TCP entirely with PCC. For example, some Content Delivery Network (CDN) providers use dedicated network infrastructure to move large amounts of data across continents [9, 10], and scientific institutes can reserve bandwidth for exchanging huge scientific data globally [26]. Second, PCC can be used in challenging network conditions **where per-user or per-tenant resource isolation is enforced** by the network. Satellite Internet providers are known to use per-user bandwidth isolation to allocate the valuable bandwidth resource [15]. For data centers with per-tenant resource isolation [30, 43, 44], an individual tenant can use PCC safely within its virtual network to address problems such as incast and improve data transfer performance between data centers.

The above applications, where PCC can fully replace or be isolated from TCP, are a significant opportunity for PCC. But in fact, **PCC does not depend on any kind of resource isolation to work.** In the public Internet, the key issue is TCP friendliness. Using PCC with the utility function described in §2.2 is not TCP friendly. However, we also study the following utility function which incorporates latency: $u_i(x) = (T_i \cdot Sigmoid_\alpha(L_i - 0.05) \cdot Sigmoid_\beta(\frac{RTT_{n-1}}{RTT_n} - 1) - x_i \cdot L_i)/RTT_n$ where RTT_{n-1} and RTT_n are the average RTT of the previous and current MI, respectively. In §4.3.1 we show that with this utility function, PCC successfully achieves TCP friendliness in various network conditions. Indeed, it is even possible for PCC to be TCP friendly while achieving much higher performance in challenging scenarios (by taking advantage of the capacity TCP's poor control algorithm leaves unused). Overall, this is a promising direction but we only take the first steps in this paper.

It is still possible that individual users will, due to its significantly improved performance, decide to deploy PCC in the public Internet with the default utility function. It turns out that the default utility function's unfriendliness to TCP is comparable to the common practice of opening parallel TCP connections used by web browsers today [3], so it is unlikely to make the ecosystem *dramatically* worse for TCP; see §4.3.2.

3 Prototype Design

We implemented a prototype of PCC in user space by adapting the UDP-based TCP skeleton in the UDT [16] package. Fig. 2 depicts our prototype's components.

3.1 Performance Monitoring

As described in §2.1 and shown in Fig. 3, the timeline is sliced into chunks of duration of T_m called the *Monitor Interval* (MI). When the Sending Module sends packets (new or retransmission) at a certain sending rate instructed by the Performance-oriented Control Module, the Monitor Module will remember what packets are sent out during each MI. As the SACK comes back from receiver, the Monitor will know what happened (received? lost? RTT?) to each packet sent out during an MI. Taking the example of Fig. 3, the Monitor knows what packets were sent during MI1, spanning T_0 to $T_0 + T_m$. At time T_1, approximately one RTT after $T_0 + T_m$, it has received the SACKs for all packets sent out in MI1. The Monitor aggregates these individual SACKs into meaningful performance metrics including throughput, loss rate and average RTT. The performance metrics are then combined by a utility function; unless otherwise stated, we use the utility function of §2.2. The result of this is that we associate a control action of each MI (sending rate) with its performance result (utility). This pair forms a "micro-experiment" and is used by the performance oriented control module.

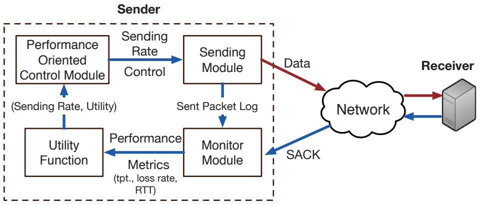


Figure 2: ***PCC prototype design***

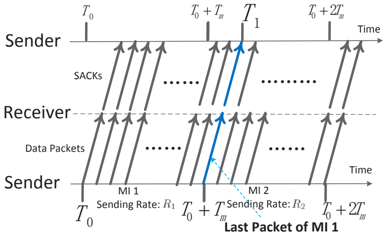


Figure 3: ***Performance monitoring process***

To ensure there are enough packets in one monitor interval, we set T_m to the maximum of (a) the time to send 10 data packets and (b) a uniform-random time in the range $[1.7, 2.2]$ RTT. Again, we want to highlight that PCC *does not* pause sending packets to wait for performance results, and it *does not* decide on a rate and send for a long time; packet transfer and measurement-control cycles occur continuously.

Note that the measurement results of one MI can arrive after the next MI has begun, and the control module can decide to change sending rate after processing this result. As an optimization, PCC will immediately change the rate and "re-align" the current MI's starting time with the time of rate change without waiting for the next MI.

3.2 Control Algorithm

We designed a practical control algorithm with the gist of the simple control algorithm described in §2.2.

Starting State: PCC starts at rate $2 \cdot MSS/RTT$ (i.e., $3KB/RTT$) and doubles its rate at each consecutive monitor interval (MI), like TCP. Unlike TCP, PCC does not exit this starting phase because of a packet loss. Instead, it monitors the utility result of each rate doubling action. Only when the utility decreases, PCC exits the starting state, returns to the previous rate which had higher utility (i.e., half of the rate), and enters the *Decision Making State*. PCC could use other more aggressive startup strategies, but such improvements could be applied to TCP as well.

Decision Making State: Assume PCC is currently at rate r. To decide which direction and amount to change its rate, PCC conducts **multiple randomized controlled trials (RCTs)**. PCC takes four consecutive MIs and divides them into two pairs (2 MIs each). For each pair, PCC attempts a slightly higher rate $r(1+\varepsilon)$ and slightly lower rate $r(1-\varepsilon)$, each for one MI, in random order. After the four consecutive trials, PCC changes the rate back to r and keeps aggregating SACKs until the Monitor generates the utility value for these four trials. For each pair $i \in 1,2$, PCC gets two utility measurements U_i^+, U_i^- corresponding to $r(1+\varepsilon), r(1-\varepsilon)$ respectively. If the higher rate consistently has higher utility ($U_i^+ > U_i^- \ \forall i \in \{1,2\}$), then PCC adjusts its sending rate to $r_{new} = r(1+\varepsilon)$; and if the lower rate consistently has higher utility then PCC picks $r_{new} = r(1-\varepsilon)$. However, if the results are inconclusive, e.g. $U_1^+ > U_1^-$ but $U_2^+ < U_2^-$, PCC stays at its current rate r and re-enters the Decision Making State with larger experiment granularity, $\varepsilon = \varepsilon + \varepsilon_{min}$. The granularity starts from ε_{min} when it enters the Decision Making State for the first time and will increase up to ε_{max} if the process continues to be inconclusive. This increase of granularity helps PCC avoid getting stuck due to noise. Unless otherwise stated, we use $\varepsilon_{min} = 0.01$ and $\varepsilon_{max} = 0.05$.

Rate Adjusting State: Assume the new rate after Decision Making is r_0 and $dir = \pm 1$ is the chosen moving direction. In each MI, PCC adjusts its rate in that direction faster and faster, setting the new rate r_n as: $r_n = r_{n-1} \cdot (1 + n \cdot \varepsilon_{min} \cdot dir)$. However, if utility falls, i.e. $U(r_n) < U(r_{n-1})$, PCC reverts its rate to r_{n-1} and moves back to the *Decision Making State*.

4 Evaluation

We demonstrate PCC's architectural advantages over the TCP family through diversified, large-scale and real-world experiments: §4.1: PCC achieves its design goal of **consistent high performance**. §4.2: PCC can actually achieve much **better fairness and convergence/stability tradeoff** than TCP. §4.3: PCC is **practically deployable** in terms of flow completion time for short flows and TCP friendliness. §4.4: PCC has a huge potential to flexibly **optimize for applications' heterogenous objectives** with fair queuing in the network rather than more complicated AQMs [47].

Transmission Pair	RTT	PCC	SABUL	CUBIC	Illinois
GPO → NYSERNet	12	818	563	129	326
GPO → Missouri	47	624	531	80.7	90.1
GPO → Illinois	35	766	664	84.5	102
NYSERNet → Missouri	47	816	662	108	109
Wisconsin → Illinois	9	801	700	547	562
GPO → Wisc.	38	783	487	79.3	120
NYSERNet → Wisc.	38	791	673	134	134
Missouri → Wisc.	21	807	698	259	262
NYSERNet → Illinois	36	808	674	141	141

Table 1: PCC significantly outperforms TCP in inter-data center environments. RTT is in msec; throughput in Mbps.

4.1 Consistent High Performance

We evaluate PCC's performance under 8 real-world challenging network scenarios with with *no algorithm tweaking for different scenarios. Unless otherwise stated, all experiments using the same default utility function of* §*2.2.* In the first 7 scenarios, PCC significantly outperforms specially engineered TCP variants.

4.1.1 Inter-Data Center Environment

Here we evaluate PCC's performance in scenarios like inter-data center data transfer [5] and dedicated CDN backbones [9] where **network resources can be isolated or reserved for a single entity**.

The GENI testbed [7], which has reservable bare-metal servers across the U.S. and reservable bandwidth [8] over the Internet2 backbone, provides us a representative evaluation environment. We choose 9 pairs of GENI sites and reserved **800Mbps** end-to-end dedicated bandwidth between each pair. We compare PCC, SABUL [29], TCP CUBIC [31] and TCP Illinois [38] over 100-second runs.

As shown in Table 1, PCC significantly outperforms TCP Illinois, by 5.2× on average and up to 7.5×. It is surprising that even in this very clean network, specially optimized TCPs still perform far from optimal. We believe some part of the gain is because the bandwidth-reserving rate limiter has a small buffer and TCP will overflow it, unnecessarily decreasing rate and also introducing latency jitter that confuses TCP Illinois. (TCP pacing will not resolve this problem; §4.1.5.) On the other hand, PCC continuously tracks the optimal sending rate by continuously measuring performance.

4.1.2 Satellite Links

Satellite Internet is widely used for critical missions such as emergency and military communication and Internet access for rural areas. Because TCP suffers from severely degraded performance on satellite links that have excessive latency (600ms to 1500ms RTT [14]) and relatively high random loss rate [42], special modifications of TCP (Hybla [23], Illinois) were proposed and special infrastructure has even been built [32, 50].

We test PCC against TCP Hybla (widely used in real-world satellite communication), Illinois and CUBIC under emulated satellite links on Emulab parameterized

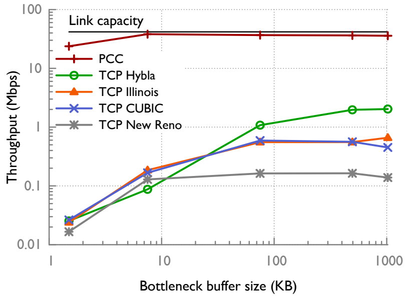


Figure 4: ***PCC outperforms special TCP modifications on emulated satellite links***

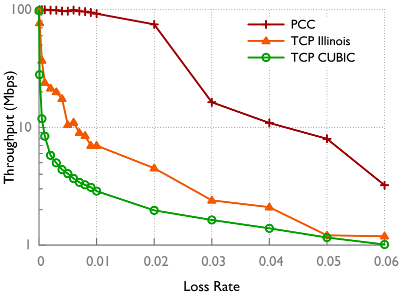


Figure 5: ***PCC is highly resilient to random loss compared to specially-engineered TCPs***

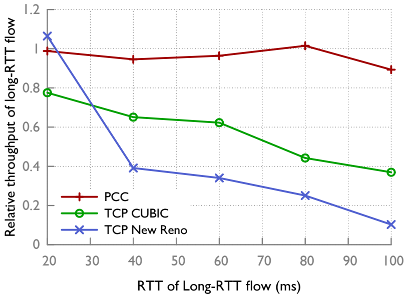


Figure 6: ***PCC achieves better RTT fairness than specially engineered TCPs***

with the real-world measurements of the WINDs satellite Internet system [42]. The satellite link has **800ms RTT, 42Mbps capacity and 0.74% random loss**. As shown in Fig. 4, we vary the bottleneck buffer from 1.5KB to 1MB and compare PCC's average throughput against different TCP variants with 100 second trials. PCC achieves 90% of optimal throughput even with only a 7.5KB buffer (5 packets) at the bottleneck. However, even with 1MB buffer, the widely used TCP Hybla can only achieve 2.03Mbps which is 17× worse than PCC. TCP Illinois, which is designed for high random loss tolerance, performs 54× worse than PCC with 1MB buffer.

4.1.3 Unreliable Lossy Links

To further quantify the effect of random loss, we set up a link on Emulab with **100Mbps bandwidth, 30ms RTT and varying loss rate**. As shown in Fig. 5, PCC can reach $>95\%$ of achievable throughput capacity until loss rate reaches 1% and shows relatively graceful performance degradation from 95% to 74% of capacity as loss rate increases to 2%. However, TCP's performance collapses very quickly: CUBIC's performance collapses to 10× smaller than PCC with only 0.1% loss rate and 37× smaller than PCC with 2% random loss. TCP Illinois shows better resilience than CUBIC but throughput still degrades severely to less than 10% of PCC's throughput with only 0.7% loss rate and 16× smaller with 2% random loss. Again, PCC can endure random loss because it looks at real utility: unless link capacity is reached, a higher rate will always result in similar loss rate and higher throughput, which translates to higher utility.

PCC's performance does decrease to 3% of the optimal achievable throughput when loss rate increases to 6% because we are using the "safe" utility function of §2.2 that caps the loss rate to 5%[1].

4.1.4 Mitigating RTT Unfairness

For unmodified TCP, short-RTT flows dominate long-RTT flows on throughput. Subsequent modifications of TCP such as CUBIC or Hybla try to mitigate this problem by making the expansion of the congestion window independent of RTT. However, the modifications cause new problems like parameter tuning (Hybla) and severely affect stability on high RTT links (CUBIC) [31]. Because PCC's convergence is based on real performance not the control cycle length, it acts as an architectural cure for the RTT unfairness problem. To demonstrate that, on Emulab we set **one short-RTT (10ms) and one long-RTT (varying from 20ms to 100ms) network path sharing the same bottleneck link of 100Mbit/s bandwidth** and buffer equal to the BDP of the short-RTT flow. We run the long-RTT flow first for 5s, letting it grab the bandwidth, and then let the short-RTT flow join to compete with the long-RTT flow for 500s and calculate the ratio of the two flows' throughput. As shown in Fig. 6, PCC achieves much better RTT fairness than New Reno and even CUBIC cannot perform as well as PCC.

4.1.5 Small Buffers on the Bottleneck Link

TCP cannot distinguish between loss due to congestion and loss simply due to buffer overflow. In face of high BDP links, a shallow-buffered router will keep chopping TCP's window in half and the recovery process is very slow. On the other hand, the practice of over-buffering networks, in the fear that an under-buffered router will drop packets or leave the network severely under-utilized, results in bufferbloat [28], increasing latency. This conflict makes choosing the right buffer size for routers a challenging multi-dimensional optimization problem [27,45,51] for network operators to balance between throughput, latency, cost of buffer memory, degree of multiplexing, etc.

Choosing the right buffer size would be much less difficult if the transport protocol could efficiently utilize a network with very shallow buffers. Therefore, we test how PCC performs with a tiny buffer and compare with TCP CUBIC, which is known to mitigate this problem. Moreover, to address the concern that the performance gain of PCC is merely due to PCC's use of packet pacing, we also test an implementation of TCP New Reno with pacing rate of $(congestionwindow)/(RTT)$. We set

[1] Throughput does not decrease to 0% because the sigmoid function is not a clean cut-off.

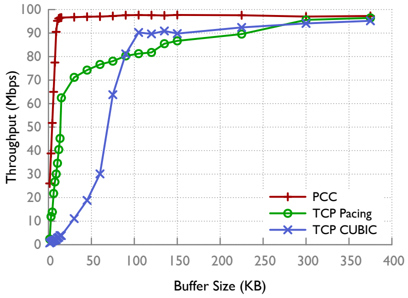


Figure 7: ***PCC efficiently utilizes shallow-buffered networks***

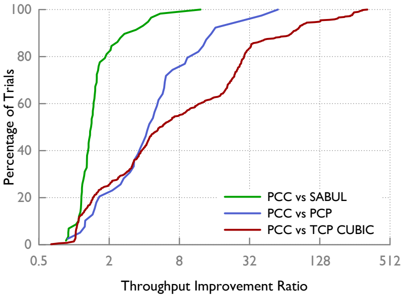


Figure 8: ***Across the public Internet, PCC has*** $\geq 10\times$ ***the performance of TCP CUBIC on*** 41% ***of tested pairs***

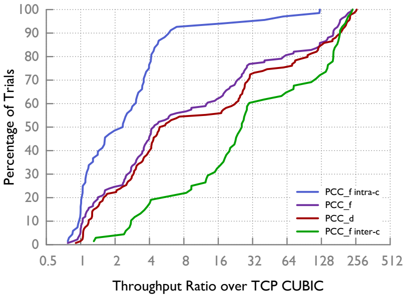


Figure 9: ***PCC's performance gain is not merely due to TCP unfriendliness***

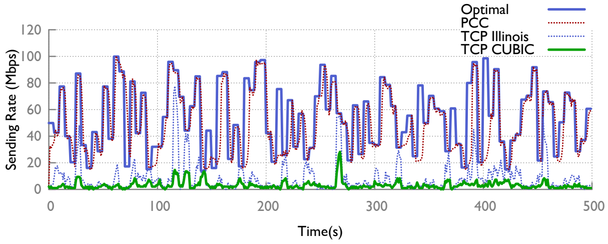


Figure 10: ***PCC can always track optimal sending rate even with drastically changing network conditions***

up on Emulab a network path with **30ms RTT, 100Mbps bottleneck bandwidth and vary the network buffer size from 1.5KB (one packet) to 375KB** ($1 \times BDP$) and compare the protocols' average throughput over 100s.

As shown in 7, PCC only requires $6 \cdot MSS$ (9 KB) buffer to reach 90% capacity. With the same buffer, CUBIC can only reach 2% capacity and even TCP with packet pacing can only reach 30%. CUBIC requires 13× more buffer than PCC to reach 90% throughput and takes 36× more buffer to close the 10% gap. Even with pacing, TCP still requires 25× more buffer than PCC to reach 90% throughput. It is also interesting to notice that with just a *single-packet* buffer, PCC's throughput can reach 25% of capacity, 35× higher throughput than TCP CUBIC. The reason is that PCC constantly monitors the real achieved performance and steadily tracks its rate at the bottleneck rate without swinging up and down like TCP. That means with PCC, network operators can use shallow buffered routers to **get low latency without harming throughput.**

4.1.6 Rapidly Changing Networks

The above scenarios are static environments. Next, we study a network that changes rapidly during the test. We set up on Emulab a network path where **available bandwidth, loss rate and RTT are all changing every** 5 **seconds.** Each parameter is chosen independently from a uniform random distribution with bandwidth ranging from 10Mbps to 100Mbps, latency from 10ms to 100ms and loss rate from 0% to 1%.

Figure 10 shows available bandwidth (optimal sending rate), and the sending rate of PCC, CUBIC and Illinois. Note that we show the PCC control algorithm's chosen sending rate (not its throughput) to get insight into how PCC handles network dynamics. Even with all network parameters rapidly changing, PCC tracks the available bandwidth very well, unlike the TCPs. Over the course of the experiment (500s), PCC's throughput averages 44.9Mbps, achieving 83% of the optimal, while TCP CUBIC and TCP Illinois are 14× and 5.6× worse than PCC respectively.

4.1.7 Big Data Transfer in the Wild

Due to its complexity, the commercial Internet is an attractive place to test whether PCC can achieve consistently high performance. We deploy PCC's receiver on 85 globally distributed PlanetLab [17] nodes and senders on 6 locations: five GENI [7] sites and our local server, and ran experiments over a 2-week period in December 2013. These 510 sending-receiving pairs render a very diverse testing environment with BDP from 14.3 KB to 18 MB.

We first test PCC against TCP CUBIC, the Linux kernel default since 2.6.19; and also SABUL [16], a special modification of TCP for high BDP links. For each sender-receiver pair, we run TCP iperf between them for 100 seconds, wait for 500 seconds and then run PCC for 100 seconds to compare their average throughput. PCC improves throughput by 5.52× at the median (Fig. 8). On 41% of sender-receiver pairs, PCC's improvement is more than 10×. This is a conservative result because 4 GENI sites have 100Mbps bandwidth limits on their Internet uplinks.

We also tested two other non-TCP transport protocols on smaller scale experiments: the public releases of PCP [12,20] (43 sending receiving pairs) and SABUL (85 sending receiving pairs). PCP uses packet-trains [33] to probe available bandwidth. However, as discussed more in §5, this bandwidth probing is different from PCC's control based on empirically observed action-utility pairs, and contains unreliable assumptions that can yield very inaccurate sample results. SABUL, widely used for scientific data transfer, packs a full set of boost-ing techniques: packet pacing, latency monitoring, random loss tolerance, etc. However, SABUL still mechanically hardwires control action to packet-level events. Fig. 8 shows PCC outperforms PCP[2] by 4.58× at the median and 15.03× at the 90th percentile, and outperforms SABUL by 1.41× at the median and 3.39× at the 90th percentile. SABUL shows an unstable control loop: it aggressively overshoots the network and then deeply falls back. On the other hand, PCC stably tracks the optimal rate. As a result, SABUL suffers from 11% loss on average compared with PCC's 3% loss.

Is PCC's performance gain merely due to TCP unfriendliness of the default utility function? In fact, PCC's high performance gain should not be surprising given our results in previous experiments, none of which involved PCC and TCP sharing bandwidth. Nevertheless, we ran another experiment, this time with PCC using the more TCP-friendly utility function described in §2.4 with $\beta = 10$ (its TCP friendliness is evaluated in § 4.3.1), from a server at UIUC[3] to 134 PlanetLab nodes in February 2015. Fig. 9 compares the results with the default utility function (PCC_d) and the friendlier utility function (PCC_f). PCC_f still shows a median of 4.38× gain over TCP while PCC_d shows 5.19×. For 50 pairs, PCC_d yields smaller than 3% higher throughput than PCC_f and for the remaining 84 pairs, the median inflation is only 14%. The use of the PCC_f utility function does not fully rule out the possibility of TCP unfriendliness, because our evaluation of its TCP friendliness (§4.3.1) does not cover all possible network scenarios involved in this experiment. However, it is highly suggestive that the performance gain is not merely due to TCP unfriendliness.

Instead, the results indicate that PCC's advantage comes from its ability to deal with complex network conditions. In particular, geolocation revealed that the large-gain results often involved cross-continent links. On cross-continent links (68 pairs), PCC_f yielded a median gain of 25× compared with 2.33× on intra-continent links (69 pairs). We believe TCP's problem with cross-continent links is not an end-host parameter tuning problem (e.g. sending/receiving buffer size), because there are paths with similar RTT where TCP can still achieve high throughput with identical OS and configuration.

4.1.8 Incast

Moving from wide-area networks to the data center, we now investigate TCP incast [24], which occurs in high bandwidth and low latency networks when multiple senders send data to one receiver concurrently, causing throughput collapse. To solve the TCP incast problem, many protocols have been proposed, including ICTCP [55] and DCTCP [18]. Here, we demonstrate PCC can achieve high performance under incast without special-purpose algorithms. We deployed PCC on Emulab [53] with **33 senders and 1 receiver**.

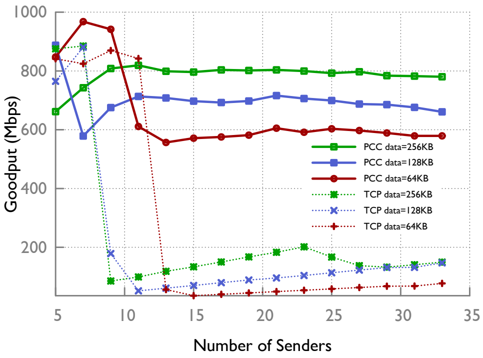
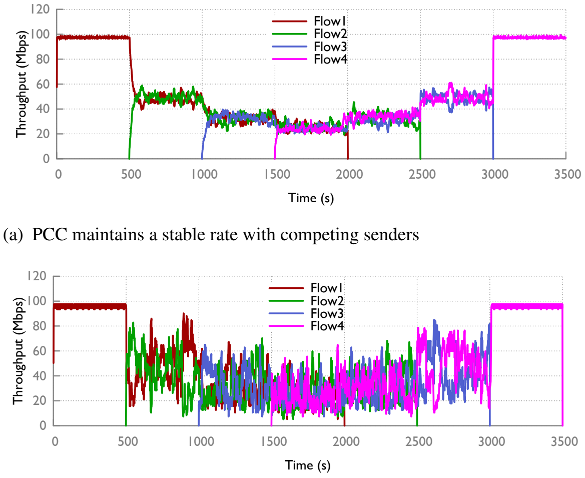


Figure 11: ***PCC largely mitigates TCP incast in a data center environment***



(a) PCC maintains a stable rate with competing senders

(b) TCP CUBIC shows high rate variance and unfairness at short time scales

Figure 12: ***PCC's dynamics are much more stable than TCP CUBIC with senders competing on a FIFO queue***

Fig. 11 shows the goodput of PCC and TCP across various flow sizes and numbers of senders. Each point is the average of 15 trials. When incast congestion begins to happen with roughly ≥ 10 senders, PCC achieves roughly 60-80% of the maximum possible goodput, or 7-8× that of TCP. Note that ICTCP [55] also achieved roughly 60-80% goodput in a similar environment. Also, DCTCP's goodput degraded with increasing number of senders [18], while PCC's is very stable.

4.2 Dynamic Behavior of Competing Flows

We proved in §2.2 that with our "safe" utility function, competing PCC flows converge to a fair equilibrium from any initial state. In this section, we experimentally show

[2] $initial-rate = 1Mbps$, $poll-interval = 100\mu s$. PCP in many cases abnormally slows down (e.g. 1 packet per 100ms). We have not determined whether this is an implementation bug in PCP or a more fundamental deficiency. To be conservative, we excluded all such results from the comparison.

[3] The OS was Fedora 21 with kernel version 3.17.4-301.

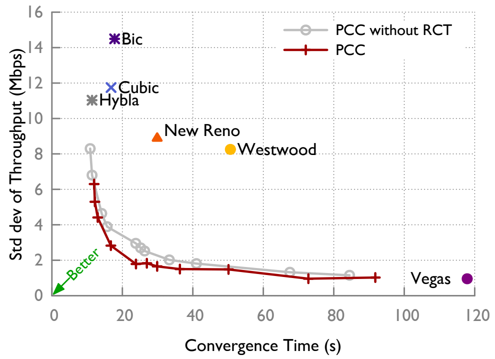


Figure 16: ***PCC has better reactiveness-stability tradeoff than TCP, particularly with its RCT mechanism***

that **PCC is much more stable, more fair and achieves a better tradeoff between stability and reactiveness than TCP.** PCC's stability can immediately translate to benefits for applications such as video streaming where stable rate in presence of congestion is desired [34].

4.2.1 PCC is More Fair and Stable Than TCP

To compare PCC and TCP's convergence process in action, we set up a dumbbell topology on Emulab with **four senders and four receivers sharing a bottleneck link with** 30**ms RTT,** 100**Mbps bandwidth**. Bottleneck router buffer size is set to the BDP to allow CUBIC to reach full throughput.

The data transmission of the four pairs initiates sequentially with a 500s interval and each pair transmits continuously for 2000s. Fig. 12 shows the rate convergence process for PCC and CUBIC respectively with 1s granularity. It is visually obvious that PCC flows converge much more stably than TCP, which has surprisingly high rate variance. Quantitatively, we compare PCC's and TCP's fairness ratio (Jain's index) at different time scales (Fig. 13). Selfishly competing PCC flows achieve better fairness than TCP at all time scales.

4.2.2 PCC has better Stability-Reactiveness tradeoff than TCP

Intuitively, PCC's control cycle is "longer" than TCP due to performance monitoring. Is PCC's significantly better stability and fairness achieved by severely sacrificing convergence time?

We set up two sending-receiving pairs sharing a bottleneck link of 100Mbps and 30ms RTT. We conduct the experiment by letting the first flow, flow A, come in the network for 20s and then let the second flow, flow B, begin. We define the convergence time in a "forward-looking" way: we say flow B's *convergence time* is the smallest t for which throughput in each second from t to $t+5s$ is within ±25% of the ideal equal rate. We measure stability by measuring the standard deviation of throughput of flow B for 60s after convergence time. All results are averaged over 15 trials. PCC can achieve various points in the stability-reactiveness trade-off space by adjusting its parameters: higher step size ε_{min} and lower monitor interval T_m result in faster convergence but higher throughput variance. In Fig. 16, we plot a trade-off curve for PCC by choosing a range of different settings of these parameters.[4] There is a clear convergence speed and stability trade-off: higher ε_{min} and lower T_m result in faster convergence and higher variance and vice versa. We also show six TCP variants as individual points in the trade-off space. The TCPs either have very long convergence time or high variance. On the other hand, PCC achieves a much better trade-off. For example, PCC with $T_m = 1.0 \cdot RTT$ and $\varepsilon_{min} = 0.02$ achieves the same convergence time and 4.2× smaller rate variance than CUBIC.

Fig. 16 also shows the **benefit of the RCT mechanism** described in §3.2. While the improvement might look small, it actually helps most in the "sweet spot" where convergence time and rate variance are both small, and where improvements are most difficult and most valuable. Intuitively, with a long monitor interval, PCC gains enough information to make a low-noise decision even in a single interval. But when it tries to make reasonably quick decisions, multiple RCTs help separate signal from noise. Though RCT doubles the time to make a decision in PCC's Decision State, the convergence time of PCC using RCT only shows slight increase because it makes *better* decisions. With $T_m = 1.0 \cdot RTT$ and $\varepsilon_{min} = 0.01$, RCT trades 3% increase in convergence time for **35% reduction in rate variance**.

4.3 PCC is Deployable

4.3.1 A Promising Solution to TCP Friendliness

		30ms	60ms	90ms
$\beta = 10$	10Mbit/s	0.94	0.75	0.67
	50Mbit/s	0.74	0.73	0.81
	90Mbit/s	0.89	0.91	1.01
$\beta = 100$	10Mbit/s	0.71	0.58	0.63
	50Mbit/s	0.56	0.58	0.54
	90Mbit/s	0.63	0.62	0.88

Table 2: ***PCC can be TCP friendly***

To illustrate that PCC does not have to be TCP unfriendly, we evaluate the utility function proposed in § 2.4. We initiate two competing flows on Emulab: a *reference flow* running TCP CUBIC, and a *competing flow* running either TCP CUBIC or PCC, under various bandwidth and latency combinations with bottleneck buffer always equal to the BDP. We compute the ratio of average (over five runs) of throughput of reference flow when it competes with CUBIC, divided by the same value when it competes with PCC. If the ratio is smaller than 1, PCC is more friendly than CUBIC. As shown in

[4] We first fix ε_{min} at 0.01 and vary the length of T_m from 4.8×RTT down to 1×RTT. Then we fix T_m at 1×RTT and vary ε_{min} from 0.01 to 0.05. This is not a full exploration of the parameter space, so other settings might actually achieve better trade-off points.

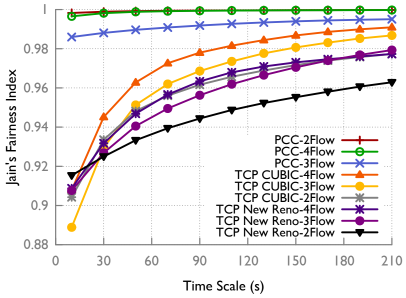


Figure 13: ***PCC achieves better fairness and convergence than TCP CUBIC***

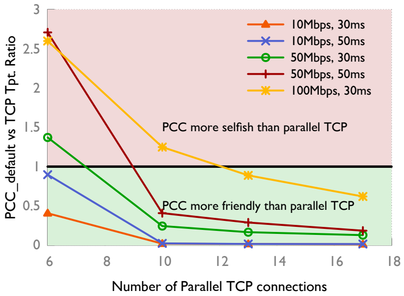


Figure 14: ***The default PCC utility function's TCP unfriendliness is similar to common selfish practice***

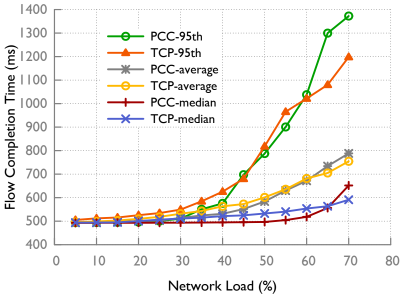


Figure 15: ***PCC can achieve flow completion time for short flows similar to TCP***

Table 2, PCC is already TCP friendly and with $\beta = 100$, PCC's performance is dominated by TCP. We consider this only a first step towards a TCP friendliness evaluation because these results also indicate PCC's friendliness can depend on the network environment. However, this initial result shows promise in finding a utility function that is sufficiently TCP friendly while also offering higher performance (note that this same utility function achieved higher performance than TCP in § 4.1.7).

4.3.2 TCP Friendliness of Default Utility Function

Applications today often adopt "selfish" practices to improve performance [3]; for example, Chrome opens between 6 (default) and 10 (maximum) parallel connections and Internet Explorer 11 opens between 13 and 17. We compare the unfriendliness of PCC's default utility function with these selfish practices by running two competing streams: one with a single PCC flow and the other with parallel TCP connections like the aforementioned web browsers. Fig. 14 shows the ratio of PCC's throughput to the total of the parallel TCP connections, over 100 seconds averaging over 5 runs under different bandwidth and RTT combinations. As shown in Fig. 14, PCC is similarly aggressive to 13 parallel TCP connections (IE11 default) and more friendly than 17 (IE11 maximum). Therefore, even using PCC's default utility function in the wild may not make the ecosystem dramatically worse for TCP. Moreover, simply using parallel connections cannot achieve consistently high performance and stability like PCC and initiating parallel connections involves added overhead in many applications.

4.3.3 Flow Completion Time for Short Flows

Will the "learning" nature of PCC harm flow completion time (FCT)? In this section, we resolve this concern by showing that with a startup phase similar to TCP (§3), PCC achieves similar FCT for short flows.

We set up a link on Emulab with 15 Mbps bandwidth and 60ms RTT. The sender sends short flows of 100KB each to receiver. The interval between two short flows is exponentially distributed with mean interval chosen to control the utilization of the link. As shown in Fig. 15, with network load ranging from 5% to 75%, PCC achieves similar FCT at the median and 95th percentile. The 95th percentile FCT with 75% utilization is 20% longer than TCP. However, we believe this is a solvable engineering issue. The purpose of this experiment is to show PCC does not fundamentally harm short flow performance. There is clearly room for improvement in the startup algorithm of all these protocols, but optimization for fast startup is intentionally outside the scope of this paper because it is a largely separate problem.

4.4 Flexiblity of PCC: An Example

In this section, we show a unique feature of PCC: expressing different data transfer objectives by using different utility functions. Because TCP is blind to the application's objective, a deep buffer (bufferbloat) is good for throughput-hungry applications but will build up long latency that kills performance of interactive applications. AQMs like CoDel [41] limits the queue to maintain low latency but degrades throughput. To cater to different applications' objective with TCP running on end hosts, it has been argued that programmable AQMs are needed in the network [47]. However, PCC can accomplish this with simple per-flow fair queuing (FQ). We only evaluate this feature in a per-flow fair queueing (FQ) environment; with a FIFO queue, the utility function may (or may not) affect dynamics and we leave that to future work. Borrowing the evaluation scenario from [47], an interactive flow is defined as a long-running flow that has the objective of maximizing its throughput-delay ratio, called the *power*. To make our point, we show that PCC + Bufferbloat + FQ has the same power for interactive flows as PCC + CoDel + FQ, and both have higher power than TCP + CoDel + FQ.

We set up a transmission pair on Emulab with 40Mbps bandwidth and 20ms RTT link running a CoDel implementation [4] with AQM parameters set to their default values. With TCP CUBIC and two simultaneous interactive flows, TCP + CoDel + FQ achieves $493.8 Mbit/s^2$, which is 10.5× more power than TCP + Bufferbloat + FQ ($46.8 Mbit/s^2$).

For PCC, we use the following utility function modified from the default to express the objective of interactive flows: $u_i(x_i) = (T_i \cdot Sigmoid(L_i - 0.05) \cdot \frac{RTT_{n-1}}{RTT_n} - x_i L_i)/RTT_n$ where RTT_{n-1} and RTT_n are the average RTT of the previous and current MIs, respectively. This utility function expresses the objective of low latency and avoiding latency increase. With this utility function, we put PCC into the same test setting of TCP. Surprisingly, PCC + Bufferbloat + FQ and PCC + CoDel + FQ achieve essentially the same power for interactive flows ($772.8 Mbit/s^2$ and $766.3 Mbit/s^2$ respectively). This is because PCC was able to keep buffers very small: we observed *no packet drop* during the experiments even with PCC + CoDel + FQ so PCC's self-inflicted latency never exceeded the latency threshold of CoDel. That is to say, CoDel becomes useless when PCC is used in end-hosts. Moreover, PCC + Bufferbloat + FQ achieves 55% higher power than TCP + CoDel + FQ, indicating that even with AQM, TCP is still suboptimal at realizing the applications' transmission objective.

5 Related work

It has long been clear that TCP lacks enough information, or the right information, to make optimal rate control decisions. XCP [35] and RCP [22] solved this by using explicit feedback from the network to directly set the sender's rate. But this requires new protocols, router hardware, and packet header formats, so deployment is rare.

Numerous designs modify TCP, e.g. [23,31,38,52,55], but fail to acheive consistent high performance, because they still inherit TCP's hardwired mapping architecture. As we evaluated in § 4, they partially mitigate TCP's problems in the specially assumed network scenarios but still suffer from performance degradation when their assumptions are violated. As another example, FAST TCP [52] uses prolonged latency as a congestion signal for high BDP connections. However, it models the network queue in an ideal way and its performance degrades under RTT variance [21], incorrect estimation of baseline RTT [49] and when competing with loss-based TCP protocols.

The method of Remy and TAO [48, 54] pushes TCP's architecture to the extreme: it searches through a large number of *hardwired mappings* under a network model with assumed parameters, e.g. number of senders, link speed, etc., and finds the best protocol under that simulated scenario. However, like all TCP variants, when the real network deviates from Remy's input assumption, performance degrades [48]. Moreover, random loss and many more real network "parameters" are not considered in Remy's network model and the effects are unclear.

Works such as PCP [20] and Packet Pair Flow Control [37] utilize techniques like packet-trains [33] to probe available bandwidth in the network. However, bandwidth probing protocols do not observe real performance like PCC does and make unreliable assumptions about the network. For example, real networks can easily violate the assumptions about packet inter-arrival latency embedded in BP (e.g. latency jitter due to middleboxes, software routers or virtualization layers), rendering incorrect estimates that harm performance.

Several past works also explicitly quantify utility. Analysis of congestion control as a global optimization [36, 40] implemented by a distributed protocol is not under the same framework as our analysis, which defines a utility function and finds the global Nash equilibrium. Other work explicitly defines a utility function for a congestion control protocol, either local [19] or global [48, 54]. However, the resulting control algorithms are still TCP-like hardwired mappings, whereas each PCC sender optimizes utility using a learning algorithm that obtains direct experimental evidence of how sending rate affects utility. Take Remy and TAO again as an example: there is a global optimization goal, used to guide the choice of protocol; but at the end of day the senders use hardwired control to attempt to optimize for that goal, which can fail when those assumptions are violated and moreover, one has to change the hardwired mapping if the goal changes.

6 Conclusion

This paper made the case that Performance-oriented Congestion Control, in which senders control their rate based on direct experimental evidence of the connection between their actions and performance, offers a promising new architecture to achieve consistent high performance. Within this architecture, many questions remain. One major area is in the choice of utility function: Is there a utility function that provably converges to a Nash equilibrium while being TCP friendly? Does a utility function which incorporates latency—clearly a generally-desirable objective—provably converge and experimentally perform as well as the default utility function used in most of our evaluation? More practically, our (userspace, UDP-based) prototype software encounters problems with accurate packet pacing and handling many flows as it scales.

7 Acknowledgement

We sincerely thank our shepherd Jon Howell and our reviewers for their valuable comments. We thank Srinivasan Keshav, Jeff Mogul, and Artur Makutunowicz for providing detailed and valuable comments at various stages of the project. We thank Google's QUIC team, including Ian Swett, for ongoing help with PCC's integration with the QUIC framework and Abdul Kabbani for helpful suggestions on software scalability. Finally, we gratefully acknowledge the support of NSF Grant 1149895, a Google Research Award, and an Alfred P. Sloan Research Fellowship.

References

[1] http://goo.gl/1Gvnis.

[2] http://www.dataexpedition.com/.

[3] Browsers usually opens parallel TCP connections to boost performance. http://goo.gl/LT0HbQ.

[4] Codel Linux implementation. http://goo.gl/06VQqG.

[5] ESNet. http://www.es.net/.

[6] Full proof of Theorem 1. http://web.engr.illinois.edu/~modong2/full_proof.pdf.

[7] GENI testbed. http://www.geni.net/.

[8] Internet2 ION service. http://webdev0.internet2.edu/ion/.

[9] Level 3 Bandwidth Optimizer. http://goo.gl/KFQ3aS.

[10] Limelight Orchestrate(TM) Content Delivery. http://goo.gl/M5oHnV.

[11] List of customers of a commercial high speed data transfer service provider. http://goo.gl/kgecRX.

[12] PCP user-space implementation. http://homes.cs.washington.edu/~arvind/pcp/pcp-ulevel.tar.gz.

[13] Report mentioning revenue range of one high speed data transfer service company. http://goo.gl/7mzB0V.

[14] Satellite link latency. http://goo.gl/xnqCih.

[15] TelliShape per-user traffic shaper. http://tinyurl.com/pm6hqrh.

[16] UDT: UDP-based data transfer. udt.sourceforge.net.

[17] PlanetLab — An open platform for developing, deploying, and accessing planetary-scale services. July 2010. http://www.planet-lab.org.

[18] M. Alizadeh, A. Greenberg, D. Maltz, J. Padhye, P. Patel, B. Prabhakar, S. Sengupta, and M. Sridharan. Data center TCP (DCTCP). *Proc. ACM SIGCOMM*, September 2010.

[19] T. Alpcan and T. Basar. A utility-based congestion control scheme for Internet-style networks with delay. *Proc. IEEE INFOCOM*, April 2003.

[20] T. Anderson, A. Collins, A. Krishnamurthy, and J. Zahorjan. PCP: efficient endpoint congestion control. *Proc. NSDI*, May 2006.

[21] H. Bullot and R. L. Cottrell. Evaluation of advanced tcp stacks on fast longdistance production networks. *Proc. PFLDNeT*, February 2004.

[22] M. Caesar, D. Caldwell, N. Feamster, J. Rexford, A. Shaikh, and K. van der Merwe. Design and implementation of a routing control platform. *Proc. NSDI*, April 2005.

[23] C. Caini and R. Firrincieli. TCP Hybla: a TCP enhancement for heterogeneous networks. *International Journal of Satellite Communications and Networking*, August 2004.

[24] Y. Chen, R. Griffith, J. Liu, R. H. Katz, and A. D. Joseph. Understanding TCP incast throughput collapse in datacenter networks. *Proc. ACM SIGCOMM Workshop on Research on Enterprise Networking*, August 2009.

[25] N. Dukkipati and N. McKeown. Why flow-completion time is the right metric for congestion control and why this means we need new algorithms. *ACM Computer Communication Review*, January 2006.

[26] ESnet. Virtual Circuits (OSCARS), May 2013. http://goo.gl/qKVOnS.

[27] Y. Ganjali and N. McKeown. Update on buffer sizing in internet routers. *ACM Computer Communication Review*, October 2006.

[28] J. Gettys and K. Nichols. Bufferbloat: Dark buffers in the Internet. *ACM Queue*, December 2011.

[29] Y. Gu, X. Hong, M. Mazzucco, and R. Grossman. SABUL: A high performance data transfer protocol. *IEEE Communications Letters*, 2003.

[30] S. Gutz, A. Story, C. Schlesinger, and N. Foster. Splendid isolation: a slice abstraction for software-defined networks. *Proc. ACM SIGCOMM Workshop on Hot Topics in Software Defined Networking*, August 2012.

[31] S. Ha, I. Rhee, and L. Xu. CUBIC: a new TCP-friendly high-speed TCP variant. *ACM SIGOPS Operating Systems Review*, 2008.

[32] Y. Hu and V. Li. Satellite-based Internet: a tutorial. *Communications Magazine*, 2001.

[33] M. Jain and C. Dovrolis. Pathload: A measurement tool for end-to-end available bandwidth. *Proc. Passive and Active Measurement (PAM)*, March 2002.

[34] J. Jiang, V. Sekar, and H. Zhang. Improving fairness, efficiency, and stability in http-based adaptive video streaming with festive. *Proc. CoNEXT*, December 2012.

[35] D. Katabi, M. Handley, and C. Rohrs. Congestion control for high bandwidth-delay product networks. *Proc. ACM SIGCOMM*, August 2002.

[36] F. Kelly, A. Maulloo, and D. Tan. Rate control for communication networks: shadow prices, proportional fairness and stability. *Journal of the Operational Research society*, 1998.

[37] S. Keshav. *The packet pair flow control protocol*. ICSI, 1991.

[38] S. Liu, T. Başar, and R. Srikant. TCP-Illinois: A loss-and delay-based congestion control algorithm for high-speed networks. *Performance Evaluation*, 2008.

[39] S. Mascolo, C. Casetti, M. Gerla, M. Sanadidi, and R. Wang. TCP Westwood: bandwidth estimation for enhanced transport over wireless links. *Proc. ACM Mobicom*, July 2001.

[40] J. Mo and J. Walrand. Fair end-to-end window-based congestion control. *IEEE/ACM Transactions on Networking (ToN)*, 2000.

[41] K. Nichols and V. Jacobson. Controlling queue delay. *Communications of the ACM*, 2012.

[42] H. Obata, K. Tamehiro, and K. Ishida. Experimental evaluation of TCP-STAR for satellite Internet over WINDS. *Proc. Autonomous Decentralized Systems (ISADS)*, June 2011.

[43] L. Popa, G. Kumar, M. Chowdhury, A. Krishnamurthy, S. Ratnasamy, and I. Stoica. Faircloud: Sharing the network in cloud computing. *Proc. ACM SIGCOMM*, August 2012.

[44] L. Popa, P. Yalagandula, S. Banerjee, J. C. Mogul, Y. Turner, and J. R. Santos. ElasticSwitch: practical

[44] L. Popa, P. Yalagandula, S. Banerjee, J. C. Mogul, Y. Turner, and J. R. Santos. ElasticSwitch: practical

work-conserving bandwidth guarantees for cloud computing. *Proc. ACM SIGCOMM*, August 2013.

[45] R. Prasad, C. Dovrolis, and M. Thottan. Router buffer sizing revisited: the role of the output/input capacity ratio. *Proc. CoNEXT*, December 2007.

[46] J. Rosen. Existence and uniqueness of equilibrium point for concave n-person games. *Econometrica*, 1965.

[47] A. Sivaraman, K. Winstein, S. Subramanian, and H. Balakrishnan. No silver bullet: extending SDN to the data plane. *Proc. HotNets*, July 2013.

[48] A. Sivaraman, K. Winstein, P. Thaker, and H. Balakrishnan. An experimental study of the learnability of congestion control. *Proc. ACM SIGCOMM*, August 2014.

[49] L. Tan, C. Yuan, and M. Zukerman. FAST TCP: Fairness and queuing issues. *IEEE Communication Letter*, August 2005.

[50] VSAT Systems. TCP/IP protocol and other applications over satellite. `http://goo.gl/E6q6Yf`.

[51] G. Vu-Brugier, R. Stanojevic, D. J. Leith, and R. Shorten. A critique of recently proposed buffer-sizing strategies. *ACM Computer Communication Review*, 2007.

[52] D. Wei, C. Jin, S. Low, and S. Hegde. Fast tcp: motivation, architecture, algorithms, performance. *IEEE/ACM Transactions on Networking*, 2006.

[53] B. White, J. Lepreau, L. Stoller, R. Ricci, G. Guruprasad, M. Newbold, M. Hibler, C. Barb, and A. Joglekar. An integrated experimental environment for distributed systems and networks. *Proc. OSDI*, December 2002.

[54] K. Winstein and H. Balakrishnan. TCP ex Machina: computer-generated congestion control. *Proc. ACM SIGCOMM*, August 2013.

[55] H. Wu, Z. Feng, C. Guo, and Y. Zhang. ICTCP: Incast congestion control for TCP in data center networks. *Proc. CoNEXT*, November 2010.

Raising the Bar for Using GPUs in Software Packet Processing

Anuj Kalia, Dong Zhou, Michael Kaminsky*, and David G. Andersen
*Carnegie Mellon University and *Intel Labs*

Abstract

Numerous recent research efforts have explored the use of Graphics Processing Units (GPUs) as accelerators for software-based routing and packet handling applications, typically demonstrating throughput several times higher than using legacy code on the CPU alone.

In this paper, we explore a new hypothesis about such designs: For many such applications, the benefits arise less from the GPU hardware itself as from the expression of the problem in a language such as CUDA or OpenCL that facilitates memory latency hiding and vectorization through massive concurrency. We demonstrate that in several cases, after applying a similar style of optimization to algorithm implementations, a CPU-only implementation is, in fact, *more resource efficient* than the version running on the GPU. To "raise the bar" for future uses of GPUs in packet processing applications, we present and evaluate a preliminary language/compiler-based framework called G-Opt that can accelerate CPU-based packet handling programs by automatically hiding memory access latency.

1 Introduction

The question of matching hardware architectures to networking requirements involves numerous trade-offs between flexibility, the use of off-the-shelf components, and speed and efficiency. ASIC implementations are fast, but relatively inflexible once designed, and must be produced in large quantities to offset the high development costs. Software routers are as flexible as code, but have comparatively poor performance, in packets-per-second (pps), as well as in cost (pps/$) and energy efficiency (pps/watt). Both ends of the spectrum are successful: Software-based firewalls are a popular use of the flexibility and affordability of systems up to a few gigabits per second; commodity Ethernet switches based on high-volume ASICs achieve seemingly unbeatable energy and cost efficiency.

In the last decade, several potential middle grounds emerged, from network forwarding engines such as the Intel IXP, to FPGA designs [12], and, as we focus on in this paper, to the use of commodity GPUs. Understanding the advantages of these architectures, and how to best exploit them, is important both in research (software-based implementations are far easier to experiment with) and in practice (software-based approaches are used for low-speed applications and in cases such as forwarding within virtual switches [13]).

Our goal in this paper is to advance understanding of the advantages of GPU-assisted packet processors compared to CPU-only designs. In particular, noting that several recent efforts have claimed that GPU-based designs can be faster even for simple applications such as IPv4 forwarding [23, 43, 31, 50, 35, 30], we attempt to identify the *reasons* for that speedup. At the outset of this work, we hypothesized that much of the advantage came from the way the GPUs were *programmed*, and that less of it came from the fundamental hardware advantages of GPUs (computational efficiency from having many processing units and huge memory bandwidth).

In this paper, we show that this hypothesis appears correct. Although GPU-based approaches are faster than a straightforward implementation of various forwarding algorithms, it is possible to transform the CPU implementations into a form that is more resource efficient than GPUs.

For many packet processing applications, the key advantage of a GPU is *not* its computational power, but that it can transparently hide the 60-200ns of latency required to retrieve data from main memory. GPUs do this by exploiting massive parallelism and using fast hardware thread switching to switch between sets of packets when one set is waiting for memory. We demonstrate that insights from code optimization techniques such as group prefetching and software pipelining [17, 51] apply to typical CPU packet handling code to boost its performance. In many cases, the CPU version is more resource efficient than the GPU, and delivers lower latency because it does not incur the additional overhead of transferring data to and from the GPU.

Finally, to make these optimizations more widely usable, both in support of practical implementations of software packet processing applications, and to give future research a stronger CPU baseline for comparison, we present a method to automatically transform data structure lookup code to overlap its memory accesses and computation. This automatically transformed code is up to 1.5-6.6x faster than the baseline code for several common

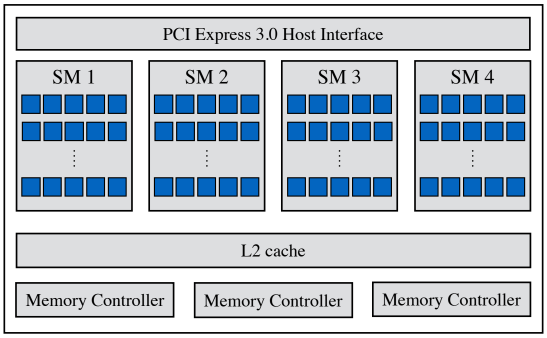


Figure 1: Simplified architecture of an NVIDIA GTX 650. The global memory is not shown.

lookup patterns, and its performance is within 10% of our hand-optimized version. By applying these optimizations, we hope to "raise the bar" for future architectural comparisons against the baseline CPU-based design.

2 Strengths and weaknesses of GPUs for packet processing

In this section, we first provide relevant background on GPU architecture and programming, and discuss the reasons why previous research efforts have used GPUs as accelerators for packet processing applications. Then, we show how the fundamental differences between the requirements of packet processing applications and conventional graphics applications make GPUs less attractive for packet processing than people often assume. Throughout this paper, we use NVIDIA and CUDA's terminology for GPU architecture and programming model, but we believe that our discussion and conclusions apply equally to other discrete GPUs (e.g., GPUs using OpenCL).

2.1 GPU strengths: vectorization and memory latency hiding

A modern CUDA-enabled GPU (Figure 1) consists of a large number of processing cores grouped into Streaming Multiprocessors (SMs). It also contains registers, a small amount of memory in a cache hierarchy, and a large global memory. The code that runs on a GPU is called a *kernel*, and is executed in groups of 32 threads called *warps*. The threads in a warp follow a SIMT (Single Instruction, Multiple Thread) model of computation: they share an instruction pointer and execute the same instructions. If the threads "diverge" (i.e., take different execution paths), the GPU selectively disables the threads as necessary to allow them to execute correctly.

Vectorization: The large number of processing cores on a GPU make it attractive as a vector processor for packets. Although network packets do have some inter-packet ordering requirements, most core networking functions such as lookups, hash computation, or encryption can be executed in parallel for multiple packets at a time. This parallelism is easily accessible to the programmer through well-established GPU-programming frameworks such as CUDA and OpenCL. The programmer writes code for a single thread; the framework automatically runs this code with multiple threads on multiple processors.

Comparison with CPUs: The AVX2 vector instruction set in the current generation of Intel processors has 256-bit registers that can process 8 32-bit integers in parallel. However, the programming language support for CPU-based vectorization is still maturing [8].

Memory latency hiding: Packet processing applications often involve lookups into large data structures kept in DRAM. Absent latency-hiding, access to these structures will stall execution while it completes (300-400 cycles for NVIDIA's GPUs). Modern GPUs hide latency using hardware. The warp scheduler in an SM holds up to 64 warps to run on its cores. When threads in a warp access global memory, the scheduler switches to a different warp. Each SM has *thousands* of registers to store the warp-execution context so that this switching does not require explicitly saving and restoring registers.

Comparison with CPUs: Three architectural features in modern CPUs enable memory latency hiding. First, CPUs have a small number of hardware threads (typically two) that can run on a single core, enabling ongoing computation when one thread is stalled on memory. Unfortunately, while each core can maintain up to ten outstanding cache misses [51], hyperthreading can only provide two "for free". Second, CPUs provide both hardware and software-managed prefetching to fetch data from DRAM into caches before it is needed. And third, after issuing a DRAM access, CPUs can continue executing independent instructions using out-of-order execution. These features, however, are less able to hide latency in unmodified code than the hardware-supported context switches on GPUs, and leave ample room for improvement using latency-hiding code optimizations (Section 3).

2.2 GPU weaknesses: setup overhead and random memory accesses

Although GPUs have attractive features for accelerating packet processing, two requirements of packet processing applications make GPUs a less attractive choice:

Many networking applications require low latency. For example, it is undesirable for a software router in a datacenter to add more than a few microseconds of latency [20]. In the measurement setup we use in this paper, the RTT through an unloaded CPU-based forwarder is 16µs. Recent work in high-performance packet processing reports numbers from 12 to 40µs [32, 51].

Unfortunately, merely communicating from the CPU to the GPU and back may add more latency than the total RTT of these existing systems. For example, it takes ∼ 15µs to transfer one byte to and from a GPU, and ~ 5µs to launch the kernel [33]. Moreover, GPU-accelerated systems must assemble large batches of packets to process on the GPU in order to take advantage of their massive parallelism and amortize setup and transfer costs. This batching further increases latency.

Networking applications often require random memory accesses into data structures, but the memory subsystem in GPUs is optimized for contiguous access. Under random accesses, GPUs lose a significant fraction of their memory bandwidth advantage over CPUs.

We now discuss these two factors in more detail. Then, keeping these two fundamental factors in mind, we perform simple experiments through which we seek to answer the following question: *When is it beneficial to offload random memory accesses or computation to a GPU?*

2.3 Experimental Setup

We perform our measurements on three CPUs and three GPUs, representing the low, mid, and high end of the recent CPU and GPU markets. Table 1 shows their relevant hardware specifications and cost. All prices are from `http://www.newegg.com` as of 9/2014. The K20 connects to an AMD Opteron 6272 socket via PCIe 2.0 x16, the GTX 980 to a Xeon E5-2680 via PCIe 2.0 x16, and the GTX 650 to an i7-4770 via PCIe 3.0 x16.

2.4 Latency of CPU-GPU communication

We first measure the minimum time required to involve a GPU in a computation—the minimum extra latency that a GPU in a software router will add to every packet. In this experiment, the host transfers an input array with N 32-bit integers to the GPU, the GPU performs negligible computations on the array, and generates an output array with the same size. To provide a fair basis for comparison with CPUs, we explored the space of possible methods for this CPU-GPU data exchange in search of the best, and present results from two methods here:

Asynchronous CUDA functions: This method performs memory copies and kernel launch using asynchronous functions (e.g., `cudaMemcpyAsync`) provided by the CUDA API. Unlike synchronous CUDA functions, these functions can reduce the total processing time by overlapping data-copying with kernel execution. Figure 2 shows the timing breakdown for the different functions. We define the time taken for an asynchronous CUDA function call as the time it takes to return control to the calling CPU thread. The extra time taken to complete all the pending asynchronous functions is shown separately.

Polling on mapped memory: To avoid the overhead of CUDA functions, we tried using CUDA's mapped memory feature that allows the GPU to access the host's memory over PCIe. We perform CPU-GPU communication using mapped memory as follows. The CPU creates the input array and a flag in the host memory and raises the flag when the input is ready. CUDA threads continuously poll the flag and read the input array when they notice a raised flag. After processing, they update the output array and start polling for the flag to be raised again. This method does not use any CUDA functions in the critical path, but all accesses to mapped memory (reading the flag, reading the input array, and writing to the output array) that come from CUDA threads lead to PCIe transactions.

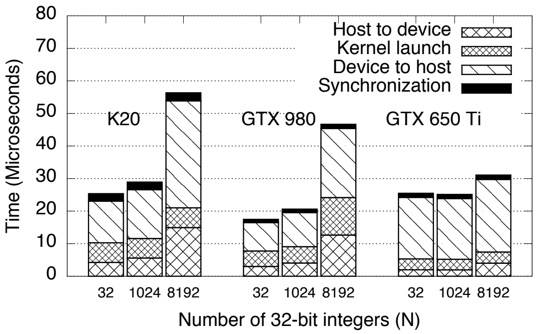


Figure 2: Timing breakdown of CPU-GPU communication with asynchronous CUDA functions.

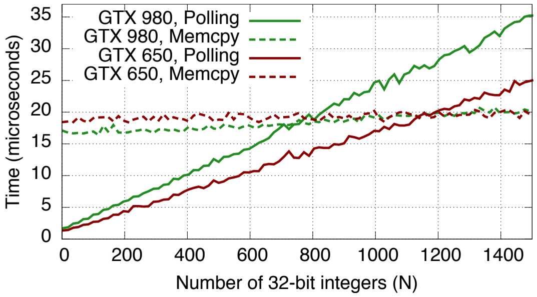


Figure 3: Minimum time for GPU-involvement with kernel-polling on mapped memory.

Figure 3 shows the time taken for this process with different values of N. The solid lines show the results with polling on mapped memory, and the dotted lines use the asynchronous CUDA functions. For small values of N, avoiding the CUDA driver overhead significantly reduces total time. However, polling generates a linearly increasing number of PCIe transactions as N increases, and becomes slower than CUDA functions for $N \sim 1000$. As GPU-offloading generally requires larger batch sizes to be efficient, we only use asynchronous CUDA functions in the rest of this work.

2.5 GPU random memory access speed

Although GPUs have much higher *sequential* memory bandwidth than CPUs (Table 1), they lose a significant fraction of their advantage when memory accesses are random, as in data structure lookups in many packet processing applications. We quantify this loss by measuring

Name	# of cores	Memory b/w	Arch., Lithography	Released	Cost	Random Access Rate
Xeon E5-2680	8	51.2 GB/s	SandyBridge, 32nm	2012	$1,748	595 M/s
Xeon E5-2650 v2	8	59.7 GB/s	IvyBridge, 22nm	2013	$1,169	464 M/s
i7-4770	4	25.6 GB/s	Haswell, 22nm	2013	$309	262 M/s
Tesla K20	2,496	208 GB/s	Kepler, 28nm	2012	$2,848	792 M/s
GTX 980	2048	224 GB/s	Maxwell, 28nm	2014	$560	1260 M/s
GTX 650 Ti	768	86.4 GB/s	Kepler, 28nm	2012	$130	597 M/s

Table 1: CPU and GPU specifications, and *measured* random access rate

the random access rate of CPUs and GPU as follows. We create a 1 GB array L containing a random permutation of $\{0,...,2^{28}-1\}$, and an array H of B random offsets into L, and pre-copy them to the GPU's memory. In the experiment, each element of H is used to follow a chain of random locations in L by executing H[i] = L[H[i]] D times. For maximum memory parallelism, each GPU thread handles one chain, whereas each CPU core handles all the chains simultaneously. Then, the random access rate is $\frac{B*D}{t}$, where t is the time taken to complete the above process.

Table 1 shows the rate achieved for different CPUs and GPUs with $D = 10$, and the value of B that gave the maximum rate ($B = 16$ for CPUs, $B = 2^{19}$ for GPUs).[1] Although the advertised memory bandwidth of a GTX 980 (224 GB/s) is 4.37x of a Xeon E5-2680, our measured random access rate is only 2.12x. This reduction in GPU bandwidth is explained by the inability of its memory controller to coalesce memory accesses done by different threads in a warp. The coalescing optimization is only done when the warp's threads access contiguous memory, which rarely happens in our experiment.

2.6 When should we offload to a GPU?

Given that involving GPUs takes several microseconds, and their random memory access rate is not much higher than that of CPUs, it is intriguing to find out in which scenarios GPU-offloading is really beneficial. Here, we focus on two widely-explored tasks from prior work: random memory accesses and expensive computations. In the rest of this paper, all experiments are done on the E5-2680 machine with the GTX 980 GPU.

2.6.1 Offloading random memory accesses

Lookups in pointer-based data structures such as IPv4/IPv6 tries and state machines follow a chain of mostly random pointers in memory. To understand the benefit of offloading these memory accesses to GPUs, we perform the experiment in Section 2.5, *but include the time taken to transfer* H *to and from the GPU*. H represents a batch of header addresses used for lookups in packet processing. We set B (the size of the batch) to 8192—slightly higher than the number of packets arriving in 100µs on our 40 Gbps network. We use different values of D, representing the variation in the number of pointer-dereferencing operations for different data structures.

Figure 4a plots the number of headers processed per second for the GPU and different numbers of CPU cores. As D increases, the overhead of the CUDA function calls gets amortized and the GPU outperforms an increasing number of CPU cores. However for $D \leq 4$, the CPU outperforms the GPU, indicating that offloading ≤ 4 dependent memory accesses (e.g., IPv4 lookups in PacketShader [23] and GALE [50]) should be slower than using the CPU only.

2.6.2 Offloading expensive computation

Although GPUs can provide substantially more computing power than CPUs, the gap decreases significantly when we take the communication overhead into account. To compare the computational power of GPUs and CPUs for varying amounts of offloaded computation, we perform a sequence of D dependent CityHash32 [4] operations on a each element of H (B is set to 8192).

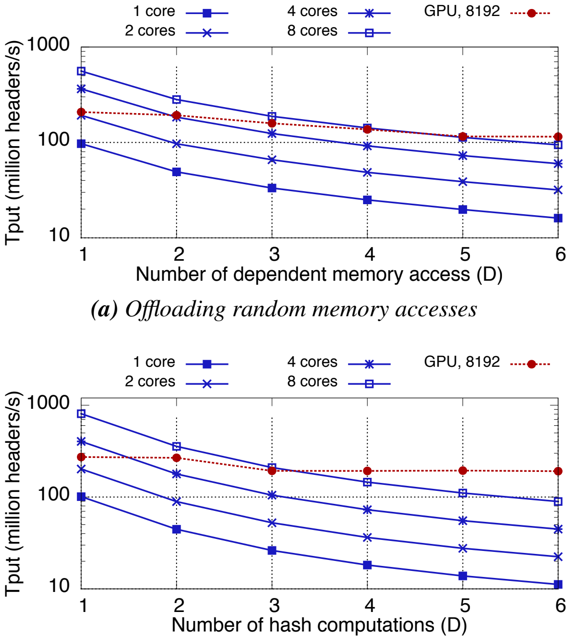
Figure 4b shows that the CPU outperforms the GPU if $D \leq 3$. Computing 3 CityHashes takes ~ 40ns on one CPU core. This time frame allows for a reasonable amount of computation before it makes sense to switch to GPU offloading. For example, a CPU core can compute the cryptographically stronger Siphash [16] of a 16-byte string in ~ 36ns.

3 Automatic DRAM latency hiding for CPUs

The section above showed that CPUs support respectable random memory access rates. However, *achieving* these rates is challenging: CPUs do not have hardware support for fast thread switching that enables latency hiding on GPUs. Furthermore, programs written for GPUs in CUDA or OpenCL start from the perspective of processing many (mostly)-independent packets, which facilitates latency hiding.

The simple experiment in the previous section saturated the CPU's random memory access capability because of its simplicity. Our code was structured such that each core issued B independent memory accesses—one for each chain—in a tight loop. The CPU has a limited window for reordering and issuing out-of-order instructions.

[1] The K20's rate increases to 1390 M/s if L is smaller than 256 MB.



(a) Offloading random memory accesses

(b) Offloading expensive computation

Figure 4: Comparison of CPU and GPU performance for commonly offloaded tasks. Note the log scale on Y axis.

When memory accesses are independent and close in the instruction stream, the CPU *can* hide the latency by issuing subsequent accesses before the first completes. However, as described below, re-structuring and optimizing real-world applications in this manner is tedious or inefficient.

A typical unoptimized packet-processing program operates by getting a batch of packets from the NIC driver, and then processing the packets one by one. Memory accesses *within* a packet are logically dependent on each other, and the memory accesses across multiple packets are spaced far apart in the instruction stream. This reduces or eliminates the memory latency-hiding effect of out-of-order execution. Our goal, then, is to (automatically) restructure this CPU code in a way that hides memory latency.

In this section, we first discuss existing techniques for optimizing CPU programs to hide their memory access latency. As these techniques are not suited to *automatically* hiding *DRAM* latency, we present a new technique called G-Opt that achieves this goal for programs with parallel data structure lookups. Although the problem of automatic parallelization and latency hiding in general is hard, certain common patterns in packet processing applications *can* be handled automatically. G-Opt hides the DRAM latency for parallel lookups that observe the same constraints as their CUDA implementations: independence across lookups and read-only data structures.

```
 1 find(entry_t *h_table, key_t *K,value_t *V) {
 2     int i;
 3     for(i = 0; i < B; i ++) {
 4         int entry_idx = hash(K[i]);
 5         // g_expensive (&h_table[entry_idx]);
 6         value_t *v_ptr = h_table[entry_idx].v_ptr;
 7         if(v_ptr != NULL) {
 8             // g_expensive (v_ptr);
 9             V[i] = *v_ptr;
10         } else {
11             V[i] = NOT_FOUND;
12         }
13     }
14 }
```

Figure 5: Naive batched hash table lookup.

3.1 Existing techniques for hiding memory access latency

3.1.1 Group prefetching

Group prefetching hides latency by processing a batch of lookups at once and by using *memory prefetches* instead of memory accesses. In a prefetch, the CPU issues a request to load a given memory location into cache, but does not wait for the request to complete. By intelligently scheduling independent instructions after a prefetch, useful work can be done while the prefetch completes. This "hiding" of prefetch latency behind independent instructions can increase performance significantly.

A data structure lookup often consists of a series of dependent memory accesses. Figure 5 shows a simple implementation of a batched hash table lookup function. Each invocation of the function processes a batch of **B** lookups. Each hash table entry contains an integer key and a pointer to a value. For simplicity, we assume for now that there are no hash collisions. There are three steps for each lookup in the batch: hash computation (line 4), accessing the hash table entry to get the value pointer (line 6), and finally accessing the value (line 9). Within a lookup, each step depends on the previous one: there are no independent instructions that can be overlapped with prefetches. However, independent instructions do exist if we consider the different lookups in the batch [17, 51].

Figure 6 is a variant of Figure 5 with the *group prefetching* optimization. It splits up the lookup code into three stages, delimited by the expensive memory accesses in the original code. We define an expensive memory access as a memory load operation that is likely to miss all levels of cache and hit DRAM. The optimized code does not directly access the hash table entry after computing the hash for a lookup key; it issues a prefetch for the entry and proceeds to compute the hash for the remaining lookups. By doing this, it does not stall on a memory lookup for the hash table entry and instead overlaps the prefetch with independent instructions (hash computation and prefetch instructions) from other lookups.

```
find(entry_t *h_table, key_t *K,value_t *V) {
    int entry_idx[B], i;
    value_t *v_ptr[B];
    // Stage 1: Hash-Computation
    for(i = 0; i < B; i ++) {
        entry_idx[i] = hash(K[i]);
        prefetch(&h_table[entry_idx[i]]);
    }

    // Stage 2: Access hash table entry
    for(i = 0; i < B; i ++) {
        v_ptr[i] = h_table[entry_idx[i]].v_ptr;
        prefetch(v_ptr[i]);
    }

    // Stage 3: Access value
    for(i = 0; i < B; i ++) {
        if(v_ptr[i] != NULL) {
            V[i] = *v_ptr[i];
        } else {
            V[i] = NOT_FOUND;
        }
    }
}
```

Figure 6: Batched lookup with group prefetching.

Unfortunately, group prefetching does not apply trivially to general lookup code because of control divergence. It requires dividing the code linearly into stages, which is difficult for code with complicated control flow. Even if such a linear layout were possible, control divergence will require a possibly large number of *masks* to record the execution paths taken by different lookups. Divergence also means that fewer lookups from a batch will enter later stages, reducing the number of instructions available to overlap with prefetches.

3.1.2 Fast context switching

In Grappa [36], fast context switching among lightweight threads is used to hide the latency of remote memory accesses over InfiniBand. After issuing a remote memory operation, the current thread yields control in an attempt to overlap the remote operation's execution with work from other threads. The minimum reported context switch time, 38 nanoseconds, is sufficiently small compared to remote memory accesses that take a few microseconds to complete. Importantly, this solution (like the hardware context switches on GPUs) is able to handle the control divergence of general packet processing. Unfortunately, the local DRAM accesses required in most packet processing applications take 60-100 nanoseconds, making the overhead of even highly optimized generic context switching unacceptable.

3.2 G-Opt

We now describe our method, called *G-Opt*, for automatically hiding DRAM latency for data structure lookup algorithms. Our technique borrows from both group prefetching and fast context switching. Individually, each of these techniques falls short of our goal: Group prefetching can hide DRAM latency but there is no general technique to automate it, and fast context switching is easy to automate but has large overhead.

G-Opt is a source-to-source transformation that operates on a batched lookup function, $\mathcal{F}$, written in C. It imposes the same constraints on the programmer that languages such as CUDA [3], OpenCL, and Intel's ISPC [8] do: the programmer must write *batch* code that expresses parallelism by granting the language explicit permission to run the code on multiple independent inputs. G-Opt additionally requires the programmer to annotate the expensive memory accesses that occur within $\mathcal{F}$. To annotate the batch lookup code in Figure 5, the lines with `g_expensive` hints should be uncommented, indicating that the following lines (line 6 and line 9) contain an expensive memory access. `g_expensive` is a macro that evaluates to an empty string: it does not affect the original code, but G-Opt uses it as a directive during code generation. The input function, $\mathcal{F}$ processes the batch of lookups one-by-one as in Figure 5. Applying G-Opt to $\mathcal{F}$ yields a new function $\mathcal{G}$ that has the same result as $\mathcal{F}$, but includes extra logic that tries to hide the latency of DRAM accesses. Before describing the transformation in more detail, we outline how the function $\mathcal{G}$ performs the lookups.

$\mathcal{G}$ begins by executing code for the first lookup. Instead of performing an expensive memory access for this lookup, $\mathcal{G}$ issues a prefetch for the access and switches to executing code for the second lookup. This continues until the second lookup encounters an expensive memory access, at which point $\mathcal{G}$ switches to the third lookup, or back to the first lookup if there are only two lookups in the batch. Upon returning to the first lookup, the new code then accesses the memory that it had previously prefetched. In the optimal case, this memory access does not need to wait on DRAM because the data is already available in the processor's L1 cache.

We now describe the transformation in more detail by discussing its action on the batched hash table lookup code in Figure 5. The code produced by G-Opt is shown in Figure 7. The key characteristics of the transformed code are:

1. **Cheap per-lookup state-maintenance**: There are two pieces of state for a lookup in $\mathcal{G}$. First, the function-specific state for a lookup is maintained in local arrays derived from the local variables in $\mathcal{F}$: the local variable named `x` in $\mathcal{F}$ is stored in `x[I]` for the `I`th lookup in $\mathcal{G}$. Second, there are two G-Opt-specific control variables for lookup `I`: `g_labels[I]` stores its `goto` target, and `g_mask`'s `I`th bit records if it has finished execution.

2. **Lookup-switching using gotos**: Instead of stalling on a memory access for lookup `I`, $\mathcal{G}$ issues a prefetch for the memory address, saves the `goto` target at the next line of code into `g_labels[I]`, and jumps to the `goto` target for the next lookup. We call this procedure a "Prefetch, Save, and Switch", or PSS. It acts as a fast switching mechanism between different lookups, and is carried out using the `G_PSS` macro that takes two arguments: the address to prefetch and the label to save as the `goto` target. G-Opt inserts a `G_PSS` macro and a `goto` target before every expensive memory access; this is achieved by using the annotations in $\mathcal{F}$.
3. **Extra initialization and termination code**: G-Opt automatically sets the initial `goto` target label for all lookups to `g_label_0`. Because different lookups can take significantly different code paths in complex applications, they can reach the label `g_end` in any order. $\mathcal{G}$ uses a bitmask to record which lookups have finished executing, and the function returns only after all lookups in the batch have reached `g_end`.

We implemented G-Opt using the ANTLR parser generator [2] framework. G-Opt performs 8 passes over the input function's Abstract Syntax Tree. It converts local variables into local arrays. It recognizes the annotations in the input function and emits labels and `G_PSS` macros. Finally, it deletes the top-level loop (written as a `foreach` loop to distinguish it from other loops) and adds the initialization and termination code based on the control variables. Our current implementation does not allow pre-processor macros in the input code, and enforces a slightly restricted subset of the ISO C grammar to avoid ambiguous cases that would normally be resolved subsequent to parsing (e.g., the original grammar can interpret `foo(x);` as a variable declaration of type `foo`).

3.3 Evaluation of G-Opt

In this section, we evaluate G-Opt on a collection of synthetic microbenchmarks that perform random memory accesses; Section 4 discusses the usefulness of G-Opt for a full-fledged software router. We present a list of our microbenchmarks along with their possible uses in real-world applications below. For each microbenchmark, we also list the source of expensive memory accesses and computation. The speedup provided by G-Opt depends on a balance between these two factors: G-Opt is not useful for compute-intensive programs with no expensive memory accesses, and loses some of its benefit for memory-intensive programs with little computation.

Cuckoo hashing: Cuckoo hashing [37] is an efficient method for storing in-memory lookup tables [19, 51]. Our 2-8 cuckoo hash table (using the terminology from MemC3 [19]) maps integer keys to integer values. *Computation*: hashing a lookup key. *Memory*: reading the corresponding entries from the hash table.

```
// Prefetch, Save label, and Switch lookup
#define G_PSS(addr, label) do {
    prefetch(addr); \
    g_labels[I] = &&label; \
    I = (I + 1) % B;  \
    goto *g_labels[I]; \
} while(0);

find(entry_t *h_table, key_t *K,value_t *V) {
    // Local variables from the function
    int entry_idx[B];
    value_t *v_ptr[B];

    // G-Opt control variables
    int I = 0, g_mask = 0;
    void *g_labels[B] = {g_label_0};

g_label_0:
    entry_idx[I] = hash(K[I]);
    G_PSS(&h_table[entry_idx[I], g_label_1);
g_label_1:
    v_ptr[I] = h_table[entry_idx[I]].v_ptr;
    if(v_ptr[I] != NULL) {
        G_PSS(v_ptr[I], g_label_2);
g_label_2:
        V[I] = *v_ptr[I];
    } else {
        V[I] = NOT_FOUND;
    }

g_end:
    g_labels[I] = &&g_end;
    g_mask = SET_BIT(g_mask, I);
    if(g_mask == (1 << B) - 1) {
        return;
    }
    I = (I + 1) % B;
    goto *g_labels[I];
}
```

Figure 7: Batched hash table lookup after G-Opt transforming.

Pointer chasing: Several algorithms that operate on pointer-based data structures, such as trees, tries, and linked lists, are based on following pointers in memory and involve little computation. We simulate a pointer-based data structure with minimal computation by using the experiment in Section 2.5. We set D to 100, emulating the long chains of dependent memory accesses performed for traversing data structures such as state machines and trees. *Computation*: negligible. *Memory*: reading an integer at a random offset in `L`.

IPv6 lookup: To demonstrate the applicability of G-Opt to real-world code, we used it to accelerate Intel DPDK's batched IPv6 lookup function. Applying G-Opt to the lookup code required only minor syntactic changes and one line of annotation, whereas hand-optimization required significant changes to the code's logic. We populated DPDK's Longest Prefix Match (LPM) structure with 200,000 random IPv6 prefixes (as done in Packet-

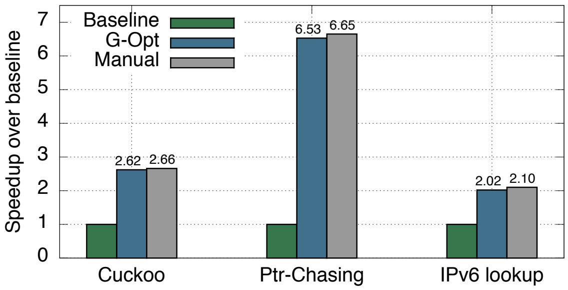


Figure 8: Speedup with G-Opt and manual group prefetching.

Shader [23]) with lengths between 48 and 64 bits,[2] and used random samples from these prefixes to simulate a worst case lookup workload. *Computation*: a few arithmetic and bitwise operations. *Memory*: 4 to 6 accesses to the LPM data structure.

Our microbenchmarks use 2 MB hugepages to reduce TLB misses [32]. We use `gcc` version 4.6.3 with -O3. The experiments in this section were performed on a Xeon E5-2680 CPU with 32 GB of RAM and 20 MB of L3 cache. We also tested G-Opt on the CPUs in Table 1 with similar results.

3.3.1 Speedup over baseline code

Figure 8 shows the benefit of G-Opt for our microbenchmarks. G-Opt speeds up cuckoo hashing by 2.6x, pointer chasing (with $D = 100$) by 6.6x, and IPv6 lookups by 2x. The figure also shows the speedup obtained by manually re-arranging the baseline code to perform group prefetching. There is modest room for further optimization of the generated code in the future, but G-Opt performs surprisingly well compared to hand-optimized code: the manually optimized code is up to 5% faster than G-Opt. For every expensive memory access, G-Opt issues a prefetch, saves a label, and executes a `goto`, but the hand-optimized code avoids the last two steps.

3.3.2 Instruction overhead of G-Opt

G-Opt's output, $\mathcal{G}$, has more code than the original input function $\mathcal{F}$. The new function needs instructions to switch between different lookups, plus the initialization and termination code. G-Opt also replaces local variable accesses with array accesses. This can lead to additional load and store instructions because array locations are not register allocated.

Although G-Opt's code executes more instructions than the baseline code, *it uses fewer cycles* by reducing the number of cycles that are spent stalled on DRAM accesses. We quantify this effect in Figure 9 by measuring the total number of instructions and the instructions-per-cycle (IPC) for the baseline and with G-Opt. We use the PAPI tool [9] to access hardware counters for total retired

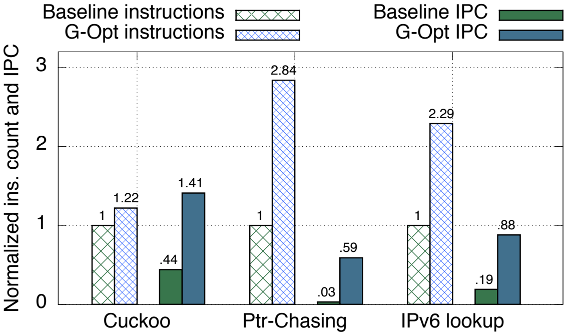


Figure 9: Instruction count and IPC with G-Opt.

instructions and total cycles. G-Opt offsets the increase in instruction count by an even larger increase in the IPC, leading to an overall decrease in execution time.

4 Evaluation

We evaluate four packet processing applications on CPUs and GPUs, each representing a different balance of computation, memory accesses, and overall processing required. We describe each application and list its computational and memory access requirements below. Although the CPU cycles used for packet header manipulation and transmission are an important source of computation, they are common to all evaluated applications and we therefore omit them from the per-application bullets. As described in Section 4.2, G-Opt also overlaps these computations with memory accesses.

Echo: To understand the limits of our hardware, we use a toy application called *Echo*. An Echo router forwards a packet to a uniformly random port P based on a random integer X in the packet's payload ($P = X \ mod \ 4$). In the GPU-offloaded version, we use the GPU to compute P from X. As this application does not involve expensive memory accesses, we do not use G-Opt on it.

IPv4 forwarding: We use Intel DPDK's implementation of the DIR-24-8-BASIC algorithm [22] for IPv4 lookups. It creates a 32 MB table for prefixes with length up to 24 bits and allocates 128 MB for longer prefixes. We populate the forwarding table with 527,961 prefixes from a BGP table snapshot [14], and use randomly generated IPv4 addresses in the workload. *Computation*: negligible. *Memory*: ~ 1 memory access on average (only 1% of our prefixes are longer than 24 bits).

IPv6 forwarding: As described in Section 3.3.

Layer-2 switch: We use the CuckooSwitch design [51]. It uses a cuckoo hash table to map MAC addresses to output ports. *Computation*: 1.5 hash-computations (on average) for determining the candidate buckets for a destination MAC address; comparing the destination MAC address with the addresses in the buckets' slots. *Memory*: 1.5 memory accesses (on average) for reading the buckets.

Named Data Networking: We use the hash-based algorithm for name lookup from Wang et al. [46], but use

[2] This prefix length distribution is close to worst case; only 1.5% and 0.1% of real-world IPv6 prefixes are longer than 48 and 64 bits, respectively [14].

cuckoo hashing instead of their more complex perfect hashing scheme. We populate our name lookup table with prefixes from a URL dataset containing 10 million URLs [45, 46]. We make two simplifications for our GPU-accelerated NDN forwarding. First, because our hash function (CityHash64) is slow on the GPU, we use a *null kernel* that does not perform NDN lookups and returns a response immediately. Second, we use fixed-size 32-byte URLs (the average URL size used in Zhang et al. [49]) in the packet headers for both CPU and GPU, generated by randomly extending or truncating the URLs from the dataset.[3]

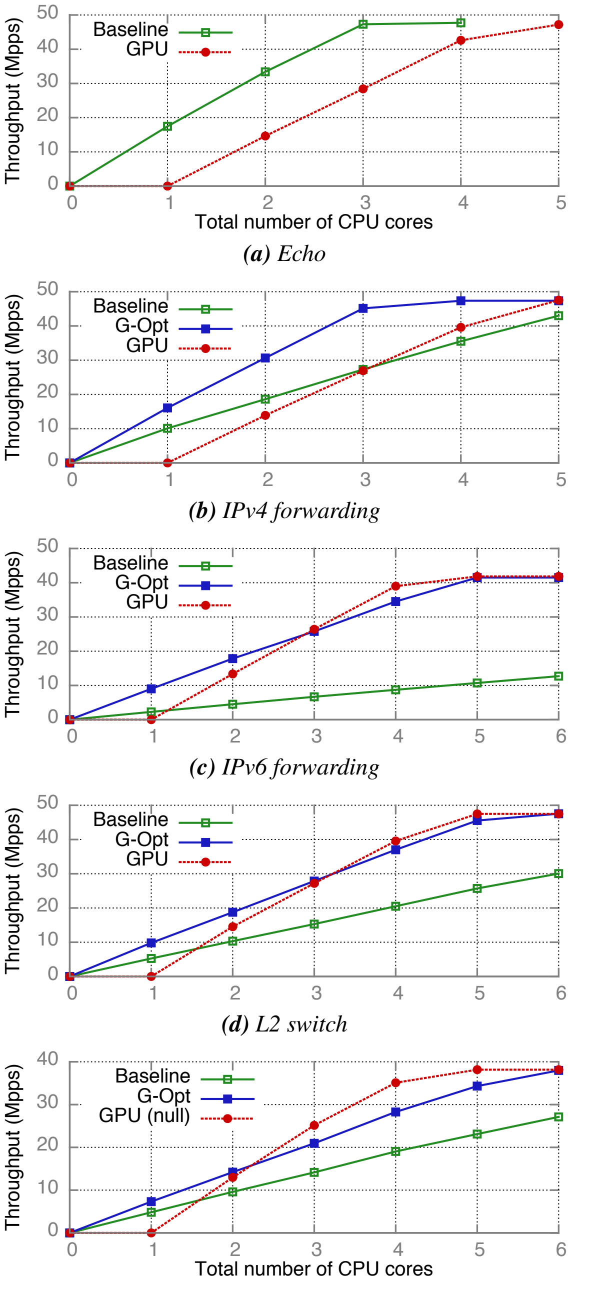
4.1 Experimental Setup

We conduct full-system experiments on a Xeon E5-2680 CPU (8 cores @2.70 GHz)-based server.[4] The CPU socket has 32 GB of quad-channel DDR3-1600 DRAM in its NUMA domain, 2 dual-port Intel X520 10 GbE NICs connected via PCIe 2.0 x8, and a GTX 980 connected via PCIe 2.0 x16. To generate the workload for the server, we use two client machines equipped with Intel L5640 CPUs (6 cores @2.27 GHz) and one Intel X520 NIC. The two 10 GbE ports on these machines are connected directly to two ports on the server. The machines run Ubuntu with Linux kernel 3.11.2 with Intel DPDK 1.5 and CUDA 6.0.

4.2 System design

Network I/O: We use Intel's DPDK [5] to access the NICs from userspace. We create as many RX and TX queues on the NIC ports as the number of active CPU cores, and ensure exclusive access to queues. Although the 40 Gbps of network bandwidth on the server machine corresponds to a maximum packet rate of 59.52 (14.88 * 4) million packets per second (Mpps) for minimum sized Ethernet frames, only 47.2 Mpps is achievable; the PCIe 2.0 x8 interface to the dual-port NIC is the bottleneck for minimum sized packets [51]. As the maximum gains from GPU acceleration come for small packets [23], we use the smallest possible packet size in all experiments.

GPU acceleration: We use PacketShader's approach to GPU-based packet processing as follows. We run a dedicated master thread that communicates with the GPU, and several worker threads that receive and transmit packets from the network. Using a single thread to communicate with the GPU is necessary because the overhead of CUDA functions increases drastically when called from multiple threads or processes. The worker threads extract the essential information from the packets and pass it on to the master thread using exclusive worker-master queues. The workers also perform standard packet processing tasks like sanity checks and setting header fields. This division of labor between workers and master reduces the amount of data that the master needs to transmit to the GPU. For example, in IPv4 forwarding, the master receives only one 4-byte IPv4 address per received packet. In our implementation, each worker can have up to 4096 outstanding packets to the master.



(a) Echo

(b) IPv4 forwarding

(c) IPv6 forwarding

(d) L2 switch

(e) Named Data Networking

Figure 10: Full system throughput with increasing *total* CPU cores, N. For the GPU, N includes the master core (throughput is zero when $N = 1$ as there are no worker cores). The x-axis label is the same for all graphs.

[3] Our CPU version does not need to make these assumptions, and performs similarly with variable length URLs.

[4] The server is dual socket, but we restricted experiments to a single CPU to avoid noise from cross-socket QPI traffic. Previous work on software packet processing suggests that performance will scale and our results will apply to two socket systems [32, 51, 23].

PacketShader's master thread issues a separate CUDA memcpy for the data generated by each worker to transfer it to the GPU directly via DMA without first copying to the master's cache. Because of the large overhead of CUDA function calls (Figure 2), we chose not to use this approach.

Using G-Opt for packet processing programs: Intel DPDK provides functions to receive and transmit batches of packets. Using batching reduces function call and PCIe transaction overheads [23, 51] and is required for achieving the peak throughput. Our baseline code works as follows. First, it calls the batched receive function to get a batch of up to 16 packets from a NIC queue. It then passes this batch to the packet processing function $\mathcal{F}$, which processes the packets one by one.

We then apply G-Opt on $\mathcal{F}$ to generate the optimized function $\mathcal{G}$. Unlike the simpler benchmarks in Section 3.3, $\mathcal{F}$ is a full-fledged packet handler: it includes code for header manipulation and packet transmission in addition to the core data structure lookup. This gives $\mathcal{G}$ freedom to overlap the prefetches from the lookups with this additional code, but also gives it permission to transmit packets in a different order than they were received. However, $\mathcal{G}$ preserves the per-flow ordering if forwarding decisions are made based on packet headers only, as in all the applications above.[5] If so, all packets from the same flow are "switched out" by $\mathcal{G}$ at the same program points, ensuring that they reach the transmission code in order.

4.3 Workload generation

The performance of the above-mentioned packet processing applications depends significantly on two workload characteristics. The following discussion focuses on IPv4 forwarding, but similar factors exist for the other applications. First, the distribution of prefixes in the server's forwarding table, and the IP addresses in the workload packets generated by the clients, affects the cache hit rate in the server. Second, in real-world traffic, packets with the same IP address (e.g., from the same TCP connection) arrive in bursts, increasing the cache hit rate.

Although these considerations are important, recall that our primary focus is understanding the relative advantages of GPU acceleration as presented in previous work. We therefore tried to mimic PacketShader's experiments that measure the near worst-case performance of both CPUs and GPUs. Thus, for IPv4 forwarding, we used a real-world forwarding table and generated the IPv4 addresses in the packets with a uniform random distribution. For IPv6 forwarding, we populated the forwarding table with prefixes with randomly generated content, and chose the workload's addresses from these prefixes using uniformly random sampling.[6] We speculate that prior work may have favored these conditions because worst-case performance is an important factor in router design for quality of service and denial-of-service resilience. Based on results from previous studies [31, 48], we also expect that more cache-friendly (non-random) workloads are likely to improve CPU performance more than that of GPUs.

4.4 Throughput comparison

Figure 10 shows the throughput of CPU-only and GPU-accelerated software routers with different numbers of CPU cores. For Echo (Figure 10a), the CPU achieves ~ 17.5 Mpps of single-core throughput and needs 3 cores to saturate the 2 dual-port 10 GbE NICs. The GPU-offloaded implementation needs at least 4 worker cores, for a total of 5 CPU cores including the master thread. This happens because the overhead of communicating each request with the master reduces the single-worker throughput to 14.6 Mpps.

Figure 10b shows the graphs for IPv4 lookup. Without G-Opt, using a GPU provides some benefit: With a budget of 4 CPU cores, the GPU-accelerated version outperforms the baseline by 12.5%. After optimizing with G-Opt, the CPU version is strictly better than the GPU-accelerated version. G-Opt achieves the platform's peak throughput with 4 CPU cores, whereas the GPU-accelerated version requires 5 CPU cores *and* a GPU.

With G-Opt, a single core can process 16 million IPv4 packets per second, which is 59% higher than the baseline's single-core performance and is only 8.9% less than the 17.5 Mpps for Echos. When using the DIR-24-8-BASIC algorithm for IPv4 lookups, the CPU needs to perform only ~ 1 expensive memory access in addition to the work done in Echo. With G-Opt, the latency of this memory access for a packet is hidden behind independent packet-handling instructions from other packets. As GPUs also hide memory access latency, the GPU-accelerated version of IPv4 forwarding performs similarly to its Echo counterpart.

For IPv6 forwarding (Figure 10c), G-Opt increases single-core throughput by 3.8x from 2.2 Mpps to 8.4 Mpps. Interestingly, this increase is *larger* than G-Opt's 2x gain in local IPv6 lookup performance (Figure 8). This counter-intuitive observation is explained by the reduction in effectiveness of the reorder buffer for the baseline code: Due to additional packet handling instructions, the independent memory accesses for different packets in a batch are spaced farther apart in the forwarding code than in the local benchmarking code. These instructions consume slots in the processor's reorder buffer, reducing its ability to detect the inter-packet independence.

[5] For applications that also examine the packet content, the transmission code can be moved outside $\mathcal{F}$ for a small performance penalty.

[6] This workload is the worst case for DPDK's trie-based IPv6 lookup. PacketShader's IPv6 lookup algorithm uses hashing and shows worst-case behavior for IPv6 addresses with random content.

With G-Opt, our CPU-only implementation achieves 39 Mpps with 5 cores, and the platform's peak IPv6 throughput (42 Mpps) with 6 cores. Because IPv6 lookups require relatively heavyweight processing, our GPU-based implementation indeed provides higher *per-worker* throughput—it delivers line rate with only 4 worker cores, *but it requires another core for the master in addition to the GPU*. Therefore, using a GPU plus 5 CPU cores can provide a 7.7% throughput increase over using just 5 CPUs, but is equivalent to using 6 CPUs.

For the L2 switch (Figure 10d), G-Opt increases the throughput of the baseline by 86%, delivering 9.8 Mpps of single-core throughput. This is significantly smaller than the 17.5 Mpps for Echos because of the expensive hash computation required by cuckoo hashing. Our CPU-only implementation saturates the NICs with 6 cores, and achieves 96% of the peak throughput with 5 cores. In comparison, our GPU-accelerated L2 switch requires 5 CPU cores and a GPU for peak throughput.

For Named Data Networking, G-Opt increases single-core throughput from 4.8 Mpps to 7.3 Mpps, a 1.5x increase. With a budget of 4 CPU cores, the (simplified) GPU version's performance is 24% higher than G-Opt, but is almost identical if G-Opt is given one additional CPU core.

Conclusion: For all our applications, the throughput gain from adding a GPU is never larger than from adding just one CPU core. The cost of a Xeon E5-2680 v3 [6] core (more powerful than the cores used in this paper) is $150. In comparison, the cheapest GPU used in this paper costs $130 and consumes 110W of extra power. CPUs are therefore a more attractive and resource efficient choice than GPUs for these applications.

4.5 Latency comparison

The GPU-accelerated versions of the above applications not only require more resources than their G-Opt counterparts, but also add significant latency. Each round of communication with the GPU on our server takes ~ 20µs (Figure 2). As the packets that arrive during a round must wait for the next round to begin, the average latency added is $20 * 1.5 = 30$µs.

Our latency experiments measure the round-trip latency at clients. Ideally, we would have liked to measure the latency *added* by the server without including the latency added by the client's NIC and network stack. This requires the use of hardware-based traffic generators [42] to which we did not have access.[7]

In our experiments, clients add a timestamp to packets during transmission and use it to measure the RTT after reception. We control the load offered by clients by tuning the amount of time they sleep between packet transmissions. The large sleep time required for generating a low load, and buffered transmission at the server [32] cause our measured latency to be higher than our system's minimum RTT of 16µs.

For brevity, we present our latency-vs-throughput graphs only for Echo, and IPv4 and IPv6 forwarding. The CPU-only versions use G-Opt. All measurements used the minimum number of CPU cores required for saturating the network bandwidth.

Figure 11a shows that the RTT of CPU-only Echo is 29µs at peak throughput and 19.5µs at low load. The minimum RTT with GPU acceleration is 52µs, which is close to 30µs larger than the CPU-only version's minimum RTT. We observe similar numbers for IPv4 and IPv6 forwarding (Figures 11b and 11c), but the GPU version's latency increases at high load because of the larger batch sizes required for efficient memory latency hiding.

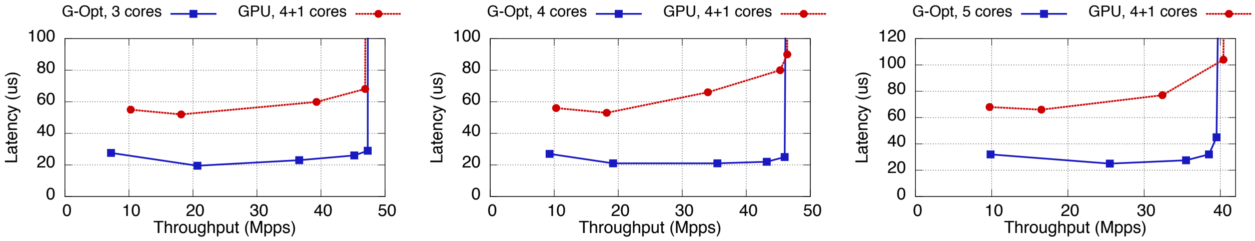
5 Discussion

Other similar optimizations for CPU programs Until now, we have discussed the benefit of an automatic DRAM latency-hiding optimization, G-Opt. We now discuss how intrusion detection systems (IDSes), *an application whose working set fits in cache* [41], can benefit from similar, latency-hiding optimizations.

We study the packet filtering stage of Snort [39], a popular IDS. In this stage, each packet's payload is used to traverse one of several Aho-Corasick [15] DFAs. The DFA represents the set of malicious patterns against which this packet should be matched; Snort chooses which DFA to use based on the packet header. For our experiments, we recorded the patterns inserted by Snort v2.9.7 into its DFAs and used them to populate our simplified pattern matching engine. Our experiment uses 23,331 patterns inserted into 450 DFAs, leading to 301,857 DFA states. The workload is a `tcpdump` file from the DARPA Intrusion Detection Data Sets [11].

Our baseline implementation of packet filtering passes batches of B (~ 8) packets to a function that returns B lists of matched patterns. This function processes packets one-by-one. We made two optimizations to this function. First, we perform a loop interchange: Instead of completing one traversal before beginning another, we interweave them to give the CPU more independent instructions to reorder, reducing stalls from long-latency loads from cache. Second, we collect a larger batch of packets (8192 in our implementation), and sort it—first by the packet's DFA number and then by length. Sorting by DFA number reduces cache misses during batch traversal. Sorting by length increases the effectiveness of loop interchange—similar to minimizing control flow divergence for GPU-based traversals [41].

[7] Experiments with a Spirent SPT-N11U [42] traffic generator as the client have measured a minimum RTT of 7-8µs on an E5-2697 v2 server; the minimum RTT measured by our clients is 16µs.



(a) *Echo* ***(b)*** *IPv4 forwarding* ***(c)*** *IPv6 forwarding*

Figure 11: Full system latency with minimum CPU cores required to saturate network bandwidth.

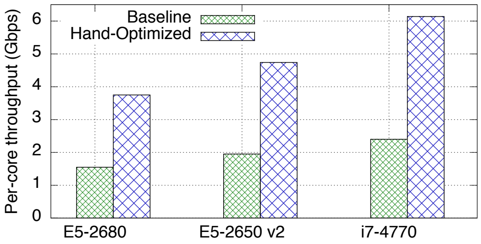


Figure 12: Pattern matching gains (no network I/O)

Figure 12 shows that, for a local experiment without network I/O, these optimizations increase single-core matching throughput by 2.4x or more. We believe that our optimizations also apply to the CPU-only versions of pattern matching in GPU-accelerated IDSes including Kargus [24] and Snap [43]. As we have only implemented the packet filtering stage (Snort uses a second, similar stage to discard false positives), we do not claim that CPUs can outperform GPUs for a full IDS. However, they can reduce the GPU advantage, or make CPU-only versions more cost effective. For example, in an experiment with innocent traffic, Kargus's throughput (with network I/O) improved between 1.4x and 2.4x with GPU offloading. Our pattern matching improvements offer similar gains which should persist in this experiment: innocent traffic rarely triggers the second stage, and network I/O requires less than 10% of the CPU cycles spent in pattern matching.

Additional applications We have shown that CPU implementations can be competitive with (or outperform) GPUs for a wide range of applications, including lightweight (IPv4 forwarding, Layer-2 switching), mid-weight (IPv6 and NDN forwarding), and heavyweight (intrusion detection) applications. Previous work explores the applicability of GPU acceleration to a number of different applications; one particularly important class, however, is cryptographic applications.

Cryptographic applications, on the one hand, involve large amounts of computation, making them seem attractive for vector processing [25, 23]. On the other hand, encryption and hashing requires copying the full packet data to the GPU (not just headers, for example). Since the publication of PacketShader, the first work in this area, Intel has implemented hardware AES encryption support for their CPUs. We therefore suspect that the 3.5x speedup observed in PacketShader for IPSec encryption would be unlikely to hold on today's CPUs. And, indeed, 6WIND's AES-NI-based IPSec implementation delivers 6 Gbps per core [1], 8x higher than PacketShader's CPU-only IPSec, though on different hardware.

One cryptographic workload where GPUs still have an advantage is processing expensive, but infrequent, RSA operations as done in SSLShader, assuming that connections arrive closely enough together for their RSA setup to be batched.[8] Being compute intensive, these cryptographic applications raise a second question for future work: Can automatic vectorization approaches (e.g., Intel's ISPC [8]) be used to increase the efficiency of CPU-based cryptographic applications?

Revising TCO estimates In light of the speedups we have shown possible for some CPU-based packet processing applications, it bears revisiting total-cost-of-ownership calculations for such machines. The TCO of a machine includes not just the cost of the CPUs, but the motherboard and chipset as well as the total system power draw, and the physical space occupied by the machine.

Although our measurements did not include power, several conclusions are obvious: Because the GPU-accelerated versions required almost as many CPU cores as the CPU-only versions, they are likely to use at least modestly more power than the CPU versions. The GTX 980 in our experiments can draw up to 165W compared to 130W for the E5-2680's 8 cores, though we lack precise power draw measurements.

Adding GPUs requires additional PCIe slots and lanes from the CPU, in addition to the cost of the GPUs. This burden is likely small for applications that require transferring only the packet header to the GPU, such as forwarding—but those applications are also a poor match for the GPU. It can, however, be significant for high-bandwidth offload applications, such as encryption and deep packet inspection.

[8] And perhaps HMAC-SHA1, but Intel's next generation "Skylake" CPUs will have hardware support for SHA-1 and SHA-256.

Future GPU trends may improve the picture. Several capabilities are on the horizon: CPU-integrated GPU functions may substantially reduce the cost of data and control transfers to the GPU. Newer NVidia GPUs support "GPUDirect" [7], which allows both the CPU and certain NICs to DMA directly packets to the GPU. GPUDirect could thus allow complete CPU-bypass from NIC to GPU, or reduce CUDA's overhead by letting the CPU write directly to GPU memory [29]. This technology currently has several restrictions—the software is nascent, and only expensive Tesla GPUs (over $1,700 each) and RDMA-capable NICs are supported. A more fundamental and long-term limitation of removing CPU involvement from packet processing is that it requires entire packets, not just headers, to be transferred to the GPU. The CPU's PCIe lanes would then have to be divided almost equally between NICs and GPUs, possibly halving the network bandwidth that the system can handle.

Alternative architectures such as Tilera's manycore designs, which place over 100 cores on a single chip with high I/O and memory bandwidth, or Intel's Xeon Phi, are interesting and under-explored possibilities. Although our results say nothing about the relative efficiency of these architectures, we hope that our techniques will enable better comparisons between them and traditional CPUs.

Handling updates Currently, G-Opt works only for data structures that are not updated concurrently. This constraint also applies to GPU-accelerated routers where the CPU constructs the data structure and ships it to the GPU. It is possible to hide DRAM latency for updates using manual group prefetching [32]; if updates are relatively infrequent, they also can be handled outside the batch lookup code. Incorporating updates into G-Opt is part of future work.

6 Related Work

GPU-based packet processing Several systems have used GPUs for IPv4 lookups *absent* network I/O [50, 31, 30, 35], demonstrating substantial speedups. Our end-to-end measurements that include network I/O, however, show that there is very little room for improving IPv4 lookup performance—when IPv4 forwarding is optimized with G-Opt, the single-core throughput drops by less than 9% relative to Echo. Packet classification requires matching packet headers against a corpus of rules; the large amount of per-packet processing makes it promising for GPU acceleration [23, 27, 44]. GSwitch [44] is a recent GPU-accelerated packet classification system. We believe that the Bloom filter and hash table lookups in GSwitch's CPU version can benefit from G-Opt's latency hiding, reducing the GPU's advantage.

CPU-based packet processing RouteBricks [18] focused on mechanisms to allocate packets to cores; its techniques are now standard for making effective use of a multicore CPU for network packet handling. User-level networking frameworks like Intel's DPDK [5], `netmap` [38], and PF_RING [10] provide a modern and efficient software basis for packet forwarding, which our work and others takes advantage of. Many of the insights in this paper were motivated by our prior work on hiding lookup latency in CuckooSwitch [51], an L2 switch that achieves 80 Gbps while storing a billion MAC addresses.

Hiding DRAM latency for CPU programs is important in many contexts: Group prefetching and software pipelining has been used to mask DRAM latency for database hash-joins [17], a software-based L2 switch [51], in-memory trees [40, 28], and in-memory key-value stores [32, 34, 26]. These systems required manual code rewrites. To our knowledge, G-Opt is the first method to automatically hide DRAM latency for the independent lookups in these applications.

7 Conclusion

Our work challenges the conclusions of prior studies about the relative performance advantages of GPUs in packet processing. GPUs achieve their parallelism and performance benefits by constraining the code that programmers can write, but this very coding paradigm also allows for latency-hiding CPU implementations. Our G-Opt tool provides a semi-automated way to produce such implementations. CPU-only implementations of IPv4, IPv6, NDN, and Layer-2 forwarding can thereby be more resource efficient and add lower latency than GPU implementations. We hope that enabling researchers and developers to more easily optimize their CPU-based designs will help improve future evaluation of both hardware and software-based approaches for packet processing. Although we have examined a wide range of applications, this work is not the end of the line. Numerous other applications have been proposed for GPU-based acceleration, and we believe that these techniques may be applicable to other domains that involve read-mostly, parallelizable processing of small requests.

Code release The code for G-Opt and the experiments in this paper is available at `https://github.com/efficient/gopt`.

Acknowledgements This work was supported by funding from the National Science Foundation under awards CNS-1314721, CCF-0964474, and 1345305; and by Intel via the Intel Science and Technology Center for Cloud Computing (ISTC-CC). Emulab [47] and PRObE [21] were used for some experiments. PRObE is supported in part by NSF awards CNS-1042537 and CNS-1042543 (PRObE). We thank Sangjin Han and Hyeontaek Lim for valuable comments, and David Maltz for shepherding.

References

[1] High-Performance Packet Processing Solutions for Intel Architecture Platforms. http://www.lannerinc.com/downloads/campaigns/LIDS/05-LIDS-Presentation-Charlie-Ashton-6WIND.pdf.

[2] ANTLR. http://www.antlr.org.

[3] NVIDIA CUDA. http://www.nvidia.com/object/cuda_home_new.html.

[4] CityHash. https://code.google.com/p/cityhash/.

[5] Intel DPDK: Data Plane Development Kit. http://dpdk.org.

[6] Intel Xeon Processor E5-2680 v3. http://ark.intel.com/products/81908/Intel-Xeon-Processor-E5-2680-v3-30M-Cache-2_50-GHz.

[7] NVIDIA GPUDirect. https://developer.nvidia.com/gpudirect.

[8] Intel's SPMD Program Compiler. https://ispc.github.io.

[9] Performance Application Programming Interface (PAPI). http://icl.cs.utk.edu/papi/.

[10] PF_RING: High-speed packet capture, filtering and analysis. http://www.ntop.org/products/pf_ring/.

[11] DARPA Intrusion Detection Data Sets, . http://www.ll.mit.edu/mission/communications/cyber/CSTcorpora/ideval/data/.

[12] NetFPGA, . http://yuba.stanford.edu/NetFPGA/.

[13] Open vSwitch, . http://www.openvswitch.org.

[14] University of Oregon Route Views Project, . http://www.routeviews.org.

[15] A. V. Aho and M. J. Corasick. Efficient String Matching: An Aid to Bibliographic Search. *Commun. ACM*, June 1975.

[16] J.-P. Aumasson and D. J. Bernstein. SipHash: a fast short-input PRF. In *INDOCRYPT*, 2012.

[17] S. Chen, A. Ailamaki, P. B. Gibbons, and T. C. Mowry. Improving Hash Join Performance Through Prefetching. *ACM Trans. Database Syst. 2007*.

[18] M. Dobrescu, N. Egi, K. Argyraki, B.-G. Chun, K. Fall, G. Iannaccone, A. Knies, M. Manesh, and S. Ratnasamy. RouteBricks: Exploiting Parallelism to Scale Software Routers. In *SOSP 2009*.

[19] B. Fan, D. G. Andersen, and M. Kaminsky. MemC3: Compact and Concurrent MemCache with Dumber Caching and Smarter Hashing. In *NSDI*, 2013.

[20] R. Gandhi, H. H. Liu, Y. C. Hu, G. Lu, J. Padhye, L. Yuan, and M. Zhang. Duet: Cloud Scale Load Balancing with Hardware and Software. In *SIGCOMM 2014*.

[21] G. Gibson, G. Grider, A. Jacobson, and W. Lloyd. PRObE: A Thousand-Node Experimental Cluster for Computer Systems Research.

[22] P. Gupta, S. Lin, and M. Nick. Routing Lookups in Hardware at Memory Access Speeds. In *INFOCOM 1998*.

[23] S. Han, K. Jang, K. Park, and S. Moon. PacketShader: A GPU-accelerated Software Router. In *SIGCOMM 2010*.

[24] M. A. Jamshed, J. Lee, S. Moon, I. Yun, D. Kim, S. Lee, Y. Yi, and K. Park. Kargus: A Highly-scalable Software-based Intrusion Detection System. In *CCS 2012*.

[25] K. Jang, S. Han, S. Han, S. Moon, and K. Park. SSLShader: Cheap SSL Acceleration with Commodity Processors. In *NSDI 2011*.

[26] A. Kalia, M. Kaminsky, and D. G. Andersen. Using RDMA Efficiently for Key-value Services. In *SIGCOMM*, 2014.

[27] K. Kang and Y. S. Deng. Scalable Packet Classification via GPU Metaprogramming. In *DATE*, 2011.

[28] C. Kim, J. Chhugani, N. Satish, E. Sedlar, A. D. Nguyen, T. Kaldewey, V. W. Lee, S. A. Brandt, and P. Dubey. Designing Fast Architecture-sensitive Tree Search on Modern Multicore/Many-core Processors. *ACM TODS 2011*, .

[29] S. Kim, S. Huh, X. Zhang, Y. Hu, A. Wated, E. Witchel, and M. Silberstein. GPUnet: Networking Abstractions for GPU Programs. In *OSDI 2014*, .

[30] T. Li, H. Chu, and P. Wang. IP Address Lookup Using GPU. In *HPSR*, 2013.

[31] Y. Li, D. Zhang, A. X. Liu, and J. Zheng. GAMT: A Fast and Scalable IP Lookup Engine for GPU-based Software Routers. In *ANCS 2013*.

[32] H. Lim, D. Han, D. G. Andersen, and M. Kaminsky. MICA: A Holistic Approach to Fast In-Memory Key-Value Storage. In *USENIX NSDI*, 2014.

[33] D. Lustig and M. Martonosi. Reducing GPU Offload Latency via Fine-grained CPU-GPU Synchronization. In *HPCA 2013*.

[34] Y. Mao, C. Cutler, and R. Morris. Optimizing RAM-latency Dominated Applications. In *APSys 2013*.

[35] S. Mu, X. Zhang, N. Zhang, J. Lu, Y. S. Deng, and S. Zhang. IP Routing Processing with Graphic Processors. In *DATE*, 2010.

[36] J. Nelson, B. Holt, B. Myers, P. Briggs, L. Ceze, S. Kahan, and M. Oskin. Grappa: A Latency-Tolerant Runtime for Large-Scale Irregular Applications. Technical Report UW-CSE-14-02-01, University of Washington.

[37] R. Pagh and F. Rodler. Cuckoo Hashing. *Journal of Algorithms*, May 2004.

[38] L. Rizzo. Netmap: A Novel Framework for Fast Packet I/O. In *USENIX ATC 2012*.

[39] M. Roesch and S. Telecommunications. Snort - Lightweight Intrusion Detection for Networks. 1999.

[40] J. Sewall, J. Chhugani, C. Kim, N. Satish, and P. Dubey. PALM: Parallel Architecture-Friendly Latch-Free Modifications to B+ Trees on Many-Core Processors. *PVLDB 2011*.

[41] R. Smith, N. Goyal, J. Ormont, K. Sankaralingam, and C. Estan. Evaluating GPUs for Network Packet Signature Matching. In *ISPASS*, 2009.

[42] Spirent. Spirent SPT-N11U. http://www.spirent.com/sitecore/content/Home/Ethernet_Testing/

Platforms/11U_Chassis.

[43] W. Sun and R. Ricci. Fast and Flexible: Parallel Packet Processing with GPUs and Click. In *ANCS 2013*.

[44] M. Varvello, R. Laufer, F. Zhang, and T. Lakshman. Multi-Layer Packet Classification with Graphics Processing Units. In *CoNEXT*, 2014.

[45] Y. Wang, Y. Zu, T. Zhang, K. Peng, Q. Dong, B. Liu, W. Meng, H. Dai, X. Tian, Z. Xu, H. Wu, and D. Yang. Wire Speed Name Lookup: A GPU-based Approach. In *NSDI 2013*.

[46] Y. Wang, B. Xu, D. Tai, J. Lu, T. Zhang, H. Dai, B. Zhang, and B. Liu. Fast name lookup for Named Data Networking. In *IWQoS*, 2014.

[47] B. White, J. Lepreau, L. Stoller, R. Ricci, S. Guruprasad, M. Newbold, M. Hibler, C. Barb, and A. Joglekar. An Integrated Experimental Environment for Distributed Systems and Networks. In *OSDI*, 2002.

[48] T. Yang, G. Xie, Y. Li, Q. Fu, A. X. Liu, Q. Li, and L. Mathy. Guarantee IP Lookup Performance with FIB Explosion. In *SIGCOMM*, 2014.

[49] T. Zhang, Y. Wang, T. Yang, J. Lu, and B. Liu. NDNBench: A benchmark for Named Data Networking lookup. In *GLOBECOM*, 2013.

[50] J. Zhao, X. Zhang, X. Wang, Y. Deng, and X. Fu. Exploiting Graphics Processors for High-performance IP Lookup in Software Routers. *INFOCOM*, 2011.

[51] D. Zhou, B. Fan, H. Lim, M. Kaminsky, and D. G. Andersen. Scalable, High Performance Ethernet Forwarding with CuckooSwitch. In *CoNEXT*, 2013.

ModNet: A modular approach to network stack extension

Sharvanath Pathak and Vivek S. Pai
Princeton University

Abstract

The existing interfaces between the network stack and the operating system are less than ideal for certain important classes of network traffic, such as video and mobile communication. While TCP has become the *de facto* transport protocol for much of this traffic, the opacity of some of the current network abstractions prevents demanding applications from controlling TCP to the fullest extent possible. At the same time, non-TCP protocols face an uphill battle as the network management and control infrastructure around TCP grows and improves.

In this paper, we introduce ModNet, a lightweight kernel mechanism that allows demanding applications better customization of the TCP stack, while preserving existing network interfaces for unmodified applications. We demonstrate ModNet's utility by implementing a range of network server enhancements for demanding environments, including adaptive bitrate video, mobile content adaptation, dynamic data and image compression, and flash crowd resource management. These enhancements operate as untrusted user-level modules, enabling easy deployment, but can still operate at scale, often providing gigabits per second of throughput with low performance overheads.

1 Introduction

With the growing popularity of HTTP, TCP has emerged as the *de facto* transport protocol for many real-time network applications (e.g. streaming video servers [7, 22]). However, since the traditional TCP stacks lack any interfaces for explicit control over the buffered content and explicit feedback about the progress of transmission, it becomes difficult to adapt quickly to changing network conditions in order to provide the desired responsiveness. As a result, web server responses are largely oblivious to network conditions, even though in the current world of mobile clients the variability of bandwidth and sudden variations in network conditions is the norm.

The lack of appropriate interfaces for exposing lower level network behavior means that applications have to rely on implicit feedback from the transmission of responses. This implicit feedback based mechanism, however, does not allow the adaptation to be performed at finer granularity. Moreover, the lack of any control over buffered data means that there can be considerable changes in network conditions between when the adaptation was performed and when the content was transmitted over the wire. Similarly, the lack of any generic interface for easily deployable, user-level customization of TCP stack behavior requires any such changes to be implemented inside the kernel, and thus, hinders their wider adoption.

One way to address these issues is to devise custom protocols, but slow adoption and difficulties with upgrading middleboxes have limited its appeal. Similarly, implementing user-level protocol stacks over raw sockets faces the compatibility restriction that some systems require superuser permission for raw socket support. The long-term issues related to developing a modified network stack or a new protocol involve the extra overhead of maintaining the stack and taking advantage of improvements in the native OS stacks. For example, UDP-based applications will often have to re-implement many common TCP behaviors to be network-friendly, and will have to develop mechanisms to interact nicely with NATs, firewalls, etc.

In this paper, we propose ModNet, a system which provides new, richer interfaces to the traditional TCP stack. The key idea behind ModNet is to loosen the boundary between network applications and the operating system and, as a result, widen the scope for enhancement of widely used network applications, such as web servers, proxy servers and multimedia streaming servers.

ModNet provides new interfaces to allow fine-grained feedback about network conditions and allows network applications greater control over buffered content. ModNet also provides an interception mechanism through *network modules* that allows easy customization of socket layer behavior. Network modules also allow application-independent deployment of new server behaviors, which would otherwise be hard-coded into specific implementations. At the same time, legacy applications remain unchanged, and the TCP stack's behavior is unchanged for these applications. We demonstrate ModNet through several modules that improve mobile content adaptation, video rebuffering and flash crowd behavior.

2 ModNet Design

The main idea behind ModNet is to give applications more insight into the operation of the network stack, and the ability to delegate management of data transfer, so that they can proactively adapt to changing network conditions. We want to loosen the boundary between applications and the network stack, so that applications or their delegates can see what is happening to the data that they want delivered, and to act on that process as the conditions of delivery change. At the same time, we want to ensure that all of the mechanisms we provide are safe, are deployable, and are maintainable.

We focus our efforts on three key areas:

- **Delegation** – allow sockets to be intercepted by one or more modules that can manipulate socket contents, parameters, and timings. These modules may be invoked directly by the application, or they may be automatically attached to application sockets by a user with the appropriate permissions. They can be composed, so that each module performs a specific task, but that a collection of modules can perform more complicated actions. In this manner, content between the application and the network can be manipulated, and the modules can be reused across applications where appropriate.

- **Inspection** – allow an easy way for interested application or modules to observe lower-level network behavior for its connections, and use this information to adapt its behavior. Normally, when an application sends data over TCP, the data is buffered and the application has no way to track its progress. We want applications to see what is happening to the data inside the network stack, so that they can react appropriately for any future data they generate.

- **Revocation** – where possible, allow applications or modules to "undo" their past behavior by modifying the contents that they have handed to the network stack, as long as such modifications do not cause any consistency problems. In practice, this means that any *unsent* data in a socket buffer can be modified if needed, allowing applications the flexibility of large socket buffers but the responsiveness of smaller ones.

In keeping with the idea that ModNet should enhance the network stack rather than disrupting it, we focus on implementing these behaviors with as few changes to the network stack as possible. Naturally, existing applications can continue to operate as usual with no modifications, but the delegation mechanism can even allow modules to operate on the network activity of otherwise unmodified network applications. We are interested in efficiency to the extent that it does not affect programmability, so that easier-to-use options are preferable to the highest possible performance. We note that CPU throughput (especially through multiple cores) has comfortably outpaced wide area network bandwidth, so maximum efficiency is not the primary driver for most networking applications. However, we take steps to ensure that ModNet is as efficient as possible within our constraints.

2.1 Delegation

In ModNet, the preferred mechanism for delegation is the use of modules, which are standalone processes that logically divert the flow of a socket between the operating system and the process that created it. These modules can be chained together, and to the application, the presence or absence of modules should be as transparent as possible.

In modern event-driven servers (e.g., Nginx) writing complex extension modules is not easy, mainly because any blocking call inside the module can block the server's event loop and thereby severely hurt the performance and scalability of the server. This possibility would get worse as modules are chained together. ModNet's delegation mechanism provides a generic extension mechanism which does not impose any such restrictions, and at the same time can be reused across applications without any extra effort. In addition, since the ModNet modules are standalone processes, they can have separate privilege levels, scheduling priorities and resource limits, which might be required for implementing critical system services.

One other alternative mechanism for implementing complex web server extensions is to use loopback proxy servers. However, proxies are not performance optimized for this usage, and are often tailored to specific application-level protocols. A performance comparison between the Nginx loopback proxy and ModNet's delegation mechanism is presented in §5.2.

We propose an interposition scheme that interposes on the sockets. This approach allows modules to examine, process, and modify the data being passed in both directions on sockets. To distinguish this approach from the existing interposition mechanisms, we term this technique *socket stealing*. This term also better describes the mechanism involved, which looks like stealing the endpoints of an existing socket and replacing them with the endpoints of the interposed module.

A schematic of the interposition mechanism for a chain of two modules (i.e., a composition of modules) is shown in Fig. 1. The application's socket $Sock_{real}$ is stolen and replaced by an *intermediate socket* $Sock_{i\_app}$, which is connected to another intermediate socket $Sock_{m\_left1}$. Since we have two modules in this case, a pair of connected intermediate sockets, namely $Sock_{m\_right1}$ and $Sock_{m\_left2}$, is created to join the two modules. To ease integration in the kernel, the sockets

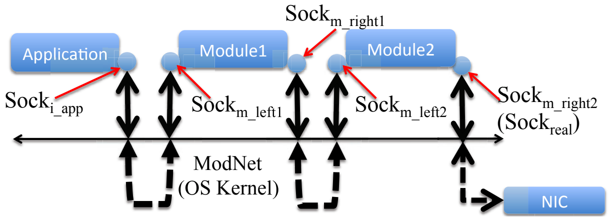


Figure 1: The architecture of ModNet's interposition mechanism for a chain of two modules. The $Sock_{real}$ is the application socket that is stolen. The black lines show the flow of data between various components.

$Sock_{m\_left1}$ and $Sock_{m\_right1}$ are mapped to the file descriptor table of the first module. Similarly, $Sock_{m\_left2}$ and $Sock_{m\_right2}$ (i.e, $Sock_{real}$) are mapped to the file descriptor table of the second module. In general, we refer to the two sockets mapped to a module's file descriptor table as $Sock_{m\_left}$ and $Sock_{m\_right}$. The modules read the data from $Sock_{m\_left}$, optionally transform it, and write the final data to $Sock_{m\_right}$, and do the same for the reverse direction. This stealing is akin to dynamically adding bidirectional pipes within an existing network connection, and this simple interface can be used to implement a large class of network functions.

2.2 Inspection

ModNet allows inspection of progress in two ways, by examining content through the interposition mechanism, and by examining the status of connections. The interposition is an *active* inspection mechanism for modules that was described in §2.1, we describe the latter now.

Adaptive network applications (e.g., adaptive video streaming servers) adapt their responses according to the network bandwidth. An interface to examine TCP state is desirable for estimating bandwidth when the clients do not explicitly provide feedback about the network conditions [17]. We propose a generic, performance efficient interface for exposing the relevant per-socket information to the applications.

ModNet allows modules and applications a fast, passive means of examining connection progress and status. The current interfaces for reading connection state for a socket (e.g., TCP state variables) are either not easy to use or are inefficient in terms of performance. For instance, in Linux one could use the tcpprobe module to read the socket state, but the interface is cumbersome, and expensive, going through the /proc pseudo-filesystem. The other alternative, the TCP_INFO socket option invokes a full system call, with no easy means of determining when an activity of interest occurs. For instance, when trying to use this status on each packet reception (e.g., in packet-pair [19] bandwidth estimation), the overhead of this polling can be significant.

The connection status tracking in ModNet extends the `mmap` (memory map) system call to socket file descriptors to implement shared readable control. The application or module receives a mapped memory region and typecasts it to the shared socket state structure. Among other fields, the shared socket state for a TCP socket contains the TCP state variables, the timestamps, sequence numbers and acknowledgment sequences for the two most recently received packets. We provide the measures for two packets because they might be needed for bandwidth estimation mechanisms such as packet-pair [19]. We use atomic reads and writes to each field in order to avoid any data races.

2.3 Revocation

The final mechanism in ModNet is for revocation, and allows the application or module to remove unsent content from socket buffers. With the growing popularity of HTTP, TCP has become the *de facto* transport protocol for many real time applications. However, the lack of interface for manipulation of socket buffer content in the current socket API makes it difficult to adapt effectively.

To better understand the need for revocation, consider the case of an HTTP-based streaming video server that handles adaptive bitrate video. It receives requests from clients for fragments of a video, using the client's estimate of what bitrate it can handle. In normal operation, the server would prefer to have very large socket buffers, so that each `write` or `sendfile` system call it performs can write the associated content to the socket buffer without blocking. If it does have to block, it would prefer to block as few times as possible, for better performance. Normal socket buffer sizes might be in the range of 16KB∼128KB or as large as 1MB for high performance servers. Moreover, socket buffers are also a form of feedback control, and the application/module may wish to monitor data transfer performance and take action when the bandwidth drops. In this case, large socket buffers lead to long delays in the control loop of the application, increasing latency, and perhaps to long rebuffering times when bandwidth drops. Small socket buffers, however, not only increase the number of system calls per piece of content, but also run the risk of not meeting client bandwidth, if they are smaller than the bandwidth-delay product or if the associated application blocks or encounters scheduling delays when the socket empties. Ideally, we want large socket buffers when things are going well, and short socket buffers when things are going poorly.

To achieve this effect, ModNet introduces a new system call, `modnet_yank`, which allows applications and modules to pull, or optionally, read a desired amount of data from the socket buffer. We describe the API details in §3.2. In the case of UDP, packets are synchronously transferred to the device queue, unless the application explicitly asks the OS to buffer them (e.g., using UDP_CORK option on Linux). Thus, modnet_yank is mostly applicable to TCP sockets. To ensure network consistency, it does not remove any data that has already been sent even if it has not been acknowledged.

3 ModNet API

ModNet has a simple and intuitive programming interface that provides better control and insight to network applications. The implementation of modules using ModNet API will be discussed in §4.2. Table 1 provides a concise overview of the ModNet API.

3.1 Delegation API

Since modules in ModNet can be standalone entities, some rendezvous mechanism is needed to have applications apply modules, and to this end, modules register themselves with the OS by name via the modnet_register system call. To steal file descriptors from a process, the module issues a modnet_getsockets system call and receives two file descriptors for each stolen socket, which correspond to the sockets $Sock_{m\_left}$ and $Sock_{m\_right}$ (see Fig. 1). If the module knows how many sockets the application will generate, it can call modnet_getsockets repeatedly, or it can register for a new EPOLL_STEAL event that ModNet delivers via the epoll event notification mechanism. Since the EPOLL_STEAL event is not tied to a file, the file descriptor argument is a negative integer denoting the CPU mask. The CPU mask is used to specify the core affinity for socket stealing, which is discussed in §4.1.

Modules can be applied individually, or in a chained fashion via the modnet_apply system call, which takes the module names and process ID. The process ID allows the application to specify that the module can be applied to itself, or to a target process, assuming they have the same owner. This mechanism can be generalized – for example, we have written a program that takes an application name and module name, and applies the module to any matching processes, or launches a new instance via fork/exec.

The modnet_yield system call allows a module to insert new modules into the chain adjacent to itself, and optionally remove itself from the chain. The system call takes two file descriptors corresponding to $Sock_{m\_left}$ and $Sock_{m\_right}$ sockets, and yields them to the specified chain of modules. The "operation" argument can be APPLY_LEFT, APPLY_RIGHT or REPLACE to instruct applying to the left of the module, right of the module or replacing the module in the chain, respectively. Naturally, the $Sock_{m\_right}$ (or $Sock_{m\_left}$) file descriptor is not required for APPLY_LEFT (or APPLY_RIGHT) operations. A usage example is the following: an HTTP filtering module that intercepts the web server connections, and based on the request invokes some other modules (e.g., Gzip compression, SSD swap, etc.) and removes itself. modnet_yield can also be used by applications (using the APPLY_RIGHT operation) to specify a chain of modules per-socket instead of using the modnet_apply, which specifies a fixed list of modules for all sockets. One can argue that the modnet_yield system call is an exhaustive interface for chain manipulation since each module is expected to be independent, and no module should modify remote portions of the chain.

3.2 Revocation API

The modnet_yank system call is used for yanking unsent content from socket buffers and modules. It can also be used for reading the existing data from socket buffers by specifying the "operation" argument as PEEK instead of YANK. Peeking can be useful in case the application wants to make the revocation decision based on the data. For instance, an HTTP streaming server might want to find a legal video frame boundary before yanking. While the application is making the decision about what to yank, it might want to prevent transmission of any new data, e.g., the streaming video server might want to prevent sending the data at the boundary of video frame. Thus, modnet_yank supports locking and the corresponding unlocking of transmission as a side effect. In order to allow this control of the application over data transmission, modnet_yank takes a "lock" argument that can be YANK_LOCK, YANK_UNLOCK or YANK_NONE.

In the case of chained modules, a call to modnet_yank might require the succeeding modules in the chain to reconstruct and return the original data. We added the EPOLL_YANK_REQ event for instructing the modules to reconstruct data. Specifically, a modnet_yank call on a $Sock_{i\_app}$ socket or an intermediate $Sock_{m\_right}$ socket leads to an EPOLL_YANK_REQ event on the succeeding module's $Sock_{m\_left}$ socket. If the succeeding module has not registered an EPOLL_YANK_REQ event on the $Sock_{m\_left}$ socket, the call returns an appropriate error (e.g., EOPNOTSUPP). On receiving the EPOLL_YANK_REQ event, the succeeding module should send back the original data via modnet_yankwrite, reconstructing it if needed. It should also perform the yank operation recursively on any successive modules. The reconstructed data is held in what we call the *yank buffer* of the socket. The data might be partially written if the yank buffer is full.

While the modules reconstruct the original data, the modnet_yank call might block or return immediately

System call	Description
modnet_register(char *mod_name)	registers the calling process using the specified module name, or fails if an existing registration has the same name. The unregister happens on the exit.
modnet_apply(pid_t target, char *mod_names[], int num_mods)	applies the named modules to the calling process or another process by pid, in the given order. Fails if either of the names do not have a corresponding active module.
modnet_getsockets(int left_fds[], int right_fds[], long cpu_mask)	gets pairs of sockets from module's steal queue and writes the two file descriptors corresponding to $Sock_{m\_left}$ and $Sock_{m\_right}$, in the array arguments, and optionally provides CPU affinity, discussed in §4.1
modnet_yield(int left_fd, int right_fd, char *mod_names[], int operation)	yields the pair of sockets with file descriptors $left\_fd$ and $right\_fd$ to the specified chain of modules. Fails if the file descriptor arguments are illegal or any of the module names do not have a corresponding active module. See §3.1 for details.
modnet_yank(int fd, void *buf, int len, int operation, int lock, int flags)	removes or copies up to the requested amount of unsent data from the socket buffer. See §3.2 for details.
modnet_yankwrite(int fd, void *buf, int len)	writes the supplied amount of data to the yank buffer of preceding socket in the chain. This system call is used by a module for returning the reconstructed data in case the preceding module/application in the chain calls a yank. See §3.2 for details.

Table 1: An overview of the ModNet API.

with an appropriate error (e.g., EAGAIN) depending on whether the YANK_DONTWAIT is set in the "flag" argument or not. The EPOLL_YANK event can be used for monitoring the readiness of yank. Note that the readiness of yank refers to the case when there is sufficient data in the yank buffer to satisfy the request.

3.3 Inspection API

The inspection API does not introduce any new system calls. The shared memory mechanism can be exposed by extending the `mmap` system call as described in §2.2. Since currently `mmap` is not supported for TCP sockets, this extension does not affect existing systems.

In the case of chained modules the `mmap` for all the intermediate sockets returns the pointer to the shared socket state of the real socket. This allows seamless composition of modules. For instance, an image compression module would read the same network state variables regardless of whether a succeeding module has been applied to it.

4 Implementation

We have implemented ModNet on Linux, modifying 364 lines of existing kernel source code and adding 2758 new lines of code to implement the new system calls and behavior. Much of the modification affects the `epoll` mechanism to support the EPOLL_STEAL event, which is tied to the current process rather than a file, and the EPOLL_YANK and EPOLL_YANK_REQ events, which take an additional length parameter. Below, we describe the implementation of the major components of ModNet.

4.1 Socket Stealing

If a module has been applied to a process, ModNet intercepts the creation of any new network sockets by the process. Two intermediate sockets are created, which correspond to $Sock_{i\_app}$ and $Sock_{m\_left}$. The $Sock_{i\_app}$ is mapped to the file descriptor table of the application and the corresponding file descriptor is returned to the application. The original socket, $Sock_{real}$, and the intermediate socket, $Sock_{m\_left}$, are added to a queue, which we call the module's *steal queue*. In case of chained modules (as in Fig. 1), a pair of connected intermediate sockets is also created for every two adjoining modules in the chain. A module reaps the entries of its *steal queue* using the `modnet_getsockets` call. The intermediate sockets are implemented by extending UNIX domain sockets.

To reduce the interposition overheads, we support batching and processor affinity for socket stealing. To amortize the costs of the `modnet_getsockets` system call, it returns the file descriptors for multiple stolen sockets in one call. Knowing that modules will often copy data, we want to allow the source and sink of the data to use the same processor cache. Borrowing the idea from Affinity Accept [24], we provide support for performing all the processing for a stolen socket on the same core. For each module, a *steal queue* is maintained per core. Any stolen sockets are enqueued to the *steal queue* of the local core. The `modnet_getsockets` call takes a bitmask as an argument called *cpu_mask*, and returns sockets only from the *steal queues* of the specified cores. While implementing the modules, we pin a thread on each core and each thread calls `modnet_getsockets`

```
epfd = epoll_create(...)
ev.events = EPOLL_STEAL
epoll_ctl(epfd, EPOLL_CTL_ADD, -mask, &ev)

while (true):
  ev = epoll_wait()
  if (ev.events & EPOLL_STEAL):
   modnet_getsockets(fd_left, fd_right, mask)
   foreach (pair of fd_left,fd_right):
    other[fd_left] = fd_right
    other[fd_right] = fd_left
    ev.events = EPOLLIN | EPOLLRDHUP
    ev.data.fd = fd_left
    epoll_add(epfd,EPOLL_CTL_ADD,fd_left,&ev)
    /* similarly add events for fd_right */
    .....
  elif (ev.events & EPOLLIN):
    /* read from ev.data.fd, and write
    to other[ev.data.fd] (omits the code
    for handling the case where the
    write buffer is full) */
    .....
  elif (ev.events & EPOLLRDHUP):
    /* signal shutdown to other end, to abide
       by protocols that are sensitive to
       end of streams (e.g. HTTP) */
    shutdown(other[ev.data.fd], SHUT_WR)
```

Figure 2: A simplified psuedo-code for the event-based bi-directional forwarder module.

with a mask that is "on" only for its local core.

To make the socket stealing mechanism as transparent to the original application as possible, we must ensure that operations intended for the original socket are actually received by the original socket. For example, if the application issues a `getpeername` system call, it would (in the absence of modules) expect to get information about the other TCP endpoint. To ensure this, all the socket system calls for the intermediate sockets, other than `recv`, `send`, `shutdown`, event notifications (`epoll`), and some socket options in `getsockopt/setsockopt` are directly translated to the corresponding $Sock_{real}$ socket. By translating a system call, we mean the effects and the return value of the system call on the intermediate socket will be identical to that of the same system call on the $Sock_{real}$ socket. The general file operations, like `close, dup, fcntl`, etc. have their usual semantics for both the intermediate sockets and the original socket. Since the stolen $Sock_{real}$ socket is the application's original socket, all the socket system calls have regular semantics for it.

4.2 Implementing Modules

Module implementations are very similar to proxies, but are simpler because they do not have to implement the whole protocol. Figure 2 shows simplified psuedo-code for a bi-directional forwarder module.

We have developed some sample modules for ModNet using the Libevent library [5] to provide scalability. A complete bi-directional forwarder module serves as the template for other modules, and is capable of handling roughly 80K connections per second (setup and teardown) with a moderately powerful 4 core server. This template is 440 lines of C code, excluding the Libevent library. Most of the code for forwarder module can directly be reused while implementing other modules. For example, the implementation of the adaptive gzip compression module §5.3 reuses this code with only 22 lines of changes.

5 Applications and Evaluation

We begin by characterizing the performance of ModNet's delegation framework through web server microbenchmarks in §5.2. The subsequent sections present evaluations using ModNet to solve some important problems for network servers for emerging classes of Internet traffic. We evaluate an adaptive gzip compression module for data (§5.3) and an adaptive JPEG compression module for images (§5.4), which handle the variable network conditions for mobile clients. We also evaluate a socket buffer swap module (§5.5), which augments the total socket buffer space by offloading part of it to SSD, and optimizes resource consumption in case of a wide spectrum of client bandwidths. §5.6 contains the evaluation of a deduplication module that handles flash crowds better by reducing duplicate buffering of content across sockets. §5.7 evaluates the use of yanking socket buffers to improve the ability of an HLS (HTTP Live Streaming [22]) server to respond to network conditions.

5.1 Experimental Setup

All the machines used in the experiments are 3.5 GHz, 4-core Intel(R) Xeon(R) E3-1270 v3 processors with 8GB of DRAM. Hyperthreading is enabled for all of the experiments. The server runs our modified Linux 3.13.5 kernel, while clients run standard Linux kernels.

Each machine has two NICs: a 10Gb NIC and a 1Gb NIC. Each machine has a 256 GB SSD drive as secondary storage, attached via a SATA-III (6Gbps) port. The SSD can sustain up to 100K IOPS and 90K IOPS, for reads and writes respectively, each of size 4KB.

We use the Linux in-kernel traffic shaper (TC) [6] for regulating link bandwidths in our experiments. To emulate the bandwidth characteristics of a real network, we use bandwidth traces of a 3G mobile network [25] in some experiments. The summary of six different traces that we use in our experiments is provided in table 2.

5.2 Overheads of Delegation

In this section, we characterize the performance overheads of ModNet's delegation framework by studying

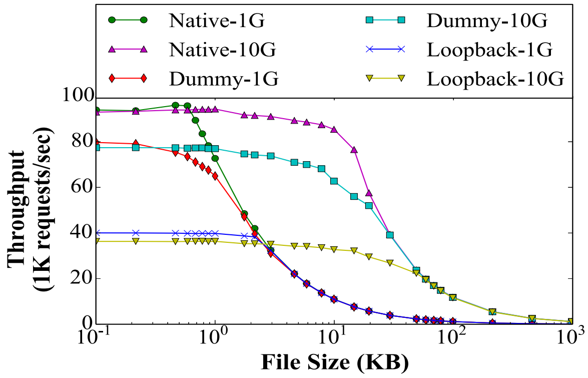


Figure 3: Nginx performance in Kreq/sec for native, loopback proxy, and a dummy Modnet module. The results are shown for both 1Gbps and 10Gbps NICs.

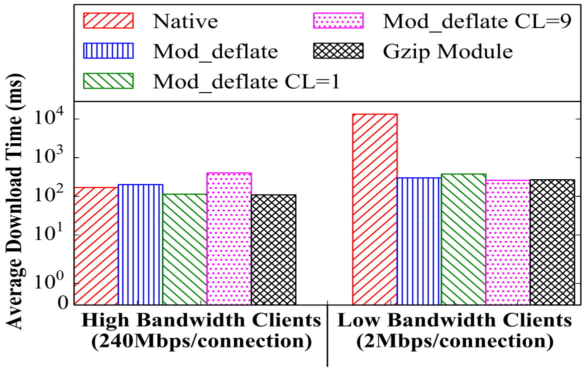


Figure 4: Performance comparison of adaptive gzip module with various configurations of mod_deflate (Apache's gzip compression implementation).

the overheads of applying a dummy module to a web server. The dummy module simply forwards data in both directions. In our micro-benchmark, a large number of clients request the same static file repeatedly, for various file sizes. The workload is CPU bound for small files and network bound for larger files.

Figure 3 shows the number of connections handled per second for a single instance of the native Nginx Web server (Native) and for the case where the dummy module is applied to it (Dummy). For comparison, we also include measurements for a loopback Nginx proxy (Loopback) applied to the Nginx instance. Experiments were performed separately for the 1Gbps and 10Gbps NICs, and the corresponding results are marked with suffixes "-1G" and "-10G", respectively, in the figures. The poor performance of the loopback proxy can be attributed to the use of IP sockets for intermediate connection between the proxy and the server, and, in general, not being as well optimized as the module implementation for this specific usage.

To generate the workload, we used two client machines running a total of 400 concurrent clients. Non-persistent connections were used in order to fully expose the per-connection overhead of the module. For small files the workload is CPU bound and ModNet poses an overhead in the range of 15-25%, which is much lower than the 50% overhead of loopback proxy. As the file size increases, the workload becomes network bound and ModNet's overhead approaches 0, after which the module provides a throughput close to the network bandwidth (i.e., ~10Gbps or 1Gbps). Inserting modules can also affect the latency of the system. In our experiments, we measured that the coefficient of variation of latencies across short connections was ~0.1 for the Native Nginx and ~0.14 for Nginx with dummy module applied to it.

In the interest of space we only discuss the conclusions from our experiment of the effect of chaining dummy modules on throughput. The throughput dropped by ~13% per additional module for a 100 byte file, and we observed 52% and 75% throughput drop for chains with 5 and 10 dummy modules, respectively. However, with the growing number of processor cores, WAN bandwidth is expected to be the bottleneck for common file sizes. Moreover, even with small files we were able to handle as much as 23K connections per second for a chain of 10 modules.

5.3 Adaptive Gzip Compression

Many web servers and proxies implement run-time content compression, so clients can still save bandwidth even when the original content was not compressed. In these cases, run-time compression can introduce extra CPU overhead, which is reasonable for slower clients, but may be a problem with fast clients or when the server CPU becomes overloaded. We use ModNet's inspection facilities to obtain fine-grained information about network conditions and adapt accordingly. Specifically, the adaptive gzip module periodically reads the socket state and drops the compression level for that transfer if its TCP congestion window is bigger than the socket buffer data, and raises the compression level otherwise. For evaluating the adaptive gzip module, we perform a similar experiment as in the last section §5.2. The number of clients was fixed to 40 for this experiment, because the compression process starts fully utilizing the CPU at that point. We used the monthly usage report of a personal Amazon EC2 account. These reports are good examples of large, dynamically generated and highly compressible content. The document size for uncompressed monthly report was 3MB and the compression ratio was in the range of 34-51.

Fig. 4 depicts the average download times for the fol-

Trace Name	Duration (s)	Mean BW (bits/s)	BW CV
Trace img1	457	2.67 M	0.54
Trace img2	390	1.95 M	0.58
Trace img3	1036	1.51 M	0.60
Trace vid1	308	730 K	0.93
Trace vid2	619	735 K	0.85
Trace vid3	430	605 K	0.95

Table 2: Total trace duration (in seconds), Mean bandwidth of the trace (BW) and the coefficient of variation (CV) of the bandwidths for various 3G bandwidth traces used in the experiments.

Website	Number of images	Total Size
BBC	24	940KB
IMDB	44	314KB
Pinterest	57	1082KB
Yahoo	20	282KB

Table 3: Characteristics of the Image datasets used.

lowing configurations: Apache without gzip compression (Native), Apache's gzip compression active with the compression level settings, default, i.e., 6 (Mod_deflate), 1 (Mod_deflate CL=1) and 9 (Mod_deflate CL=9), Apache with gzip compression disabled and our adaptive gzip module applied to it (Gzip Module). Note that the Y-axis is in log scale to accommodate large range of values.

To demonstrate the benefit of the adaptive behavior, we perform experiments with two classes of clients: (1) high bandwidth clients, where each client has a bandwidth of 240Mbps, and (2) low bandwidth clients, where each client has a bandwidth of 2Mbps, which is around the median of the mobile bandwidth samples we used. As shown in the figure, for high bandwidth clients, the bottleneck is compression speed, and thus, compression level 1 is almost 4X faster than compression level 9, 2X faster than the default case and 1.5X faster than no compression. However, for low bandwidth clients, network bandwidth is the bottleneck, so the compression level 9 is almost 1.5X faster than compression level 1, 1.2X faster than the default compression level and 51X faster than no compression. The adaptive gzip module gives the optimal performance for both cases. Although this experiment is designed to to illustrate the benefits of the inspection mechanism, it is worth noticing that even with the high bandwidth of 240Mbps (i.e., 40 clients on a 10Gbps uplink) ModNet's modularization overhead does not have any visible effect on performance relative to that of the compression process, while we get extra flexibility by using a separate network module.

5.4 Adaptive Image Compression

Given the enormous variability in bandwidth of clients in the current internet, and the fact that more than 60% [2] of the transferred bytes for an average webpage are images, serving images at a fixed resolution may be suboptimal. Even serving image resolution based on device type may be problematic, since smartphones with wi-fi access may be faster than desktops using dial-up.

An adaptive approach could select from multiple statically compressed variants of the same image or could dynamically re-compress the images. We used dynamic re-compression of images because it allows us to change the compression level on the fly based on a passive bandwidth estimate that is acquired as the connection progresses. Moreover, dynamic re-compression is suitable for transformational proxies, which have been argued to be better for incremental deployment and amortization of operating costs [14] (Google has already deployed a compression proxy that dynamically re-encodes images [2] and other content for the chrome browser).

We employed ModNet's inspection mechanism in order to obtain a fine-grained estimate of the client's bandwidth. We use a passive bandwidth estimation mechanism since it allows us to work with unmodified clients. Our estimation mechanism is based on the packet-pair [19] estimation, where we consider the pair of last two acknowledged packets if they were sent close enough and have similar sizes. Our bandwidth estimation mechanism works well for HTTP transfers. We use ModNet's delegation framework for implementing the adaptive JPEG module. This recompression can be performed at a server or at a performance-enhancing middlebox [33].

We use the JPEG image format for this module, since it is widely used and supported across browsers. Changing the image compression dynamically for JPEG images is, however, not straightforward, because as per the JPEG specification [4], there is a single quantization matrix for each color component, and it precedes the whole scan data. We devise a new scheme where we zero out the higher coefficients to get a better run length encoding (RLE), and thus a better compression ratio.

Fig. 5 and Fig. 6 show the total download time and average image quality for the four image datasets, for the native Nginx web server (Native) and with our adaptive JPEG module applied to it (JPEG Module). We used the SSIM index [32] for estimating the image quality. The bandwidth is being shaped according to the 3G network traces (see Table 2 for details). These results suggest that the adaptive JPEG module keeps the download times reasonable, regardless of how poor the client's network bandwidth is. It is worth mentioning here that the trade-off between the reduction in size vs. degradation in image quality is a policy question, and the results are shown for one specific policy that we used. For this experiment, we used only one active client. We now study

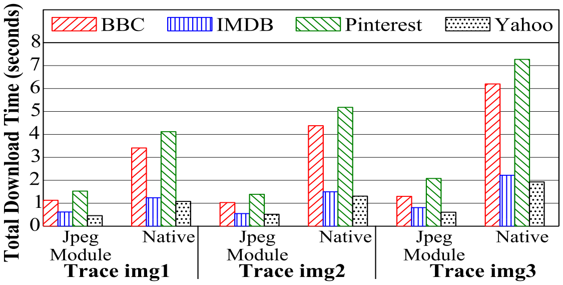


Figure 5: The load times for Nginx and Nginx with the adaptive JPEG module.

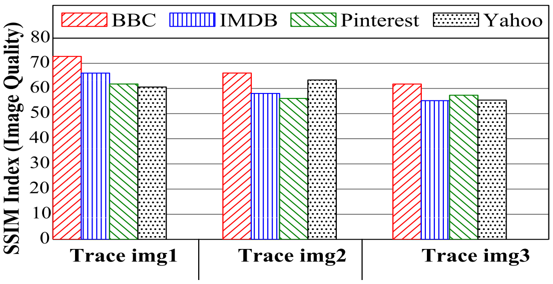


Figure 6: The average quality estimates (SSIM indices) for Nginx with the adaptive JPEG module.

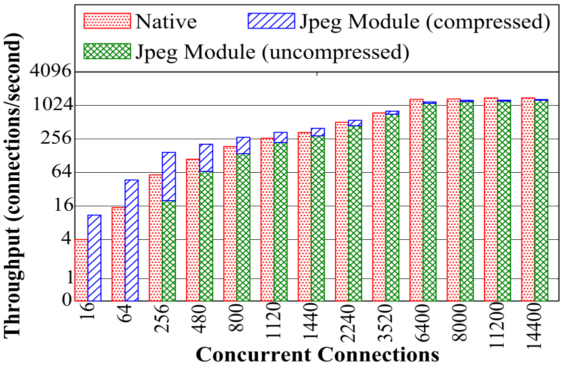


Figure 7: The throughput of Nginx with and without the adaptive JPEG module for the BBC image dataset. Note that the Y axis is in log scale.

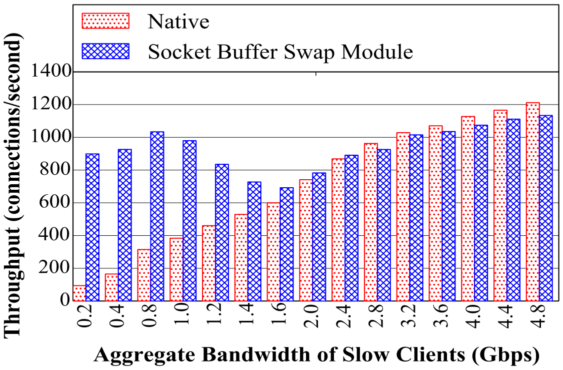


Figure 8: Performance comparison of dynamic content serviced by Apache with and without the socket buffer swap module.

the effect of concurrent connections.

For a large number of concurrent clients, the computational cost of re-encoding images becomes a matter of concern. However, since we only change the coefficients, we only have to handle the RLE step, and not the more expensive DCT (discrete cosine transform) processing. We estimated that doing an inverse DCT and a DCT for each image would require almost 3 times more computation.

To provide consistent throughput, the module only compresses a fraction of images if the CPU becomes the bottleneck. The throughput results for the adaptive JPEG module are shown in Fig. 7. The *JPEG Module (compressed)* and *JPEG Module (uncompressed)* correspond to the fraction of connections being serviced as compressed and uncompressed images, respectively, with the adaptive JPEG module applied to Nginx. Each connection involves a request for all the images in the BBC image dataset. The “Trace img2” (ref. table 2 for details) bandwidth trace is used for this experiment. The adaptive JPEG compression module provides up to 3 times more request throughput when the server is not heavily loaded.

5.5 Swappable Socket Buffers

To demonstrate the flexibility of the ModNet approach, we demonstrate its behavior in managing network socket buffer space, irrespective of usage. A large socket buffer can increase performance by reducing the chances of an application getting blocked on socket writes. As an example, consider an Apache server handling PHP requests, which use a separate process per connection. These systems typically cap the number of processes to avoid overloading the server, but if too many slow clients access the server, the PHP processes may be blocked on writing to the clients, even if the responses have been generated. Increasing the socket buffer size can reduce this chance and free the application resources early, at the expense of increasing kernel memory consumption.

One solution to this problem is to swap socket buffers that are being drained too slowly, which reduces kernel memory usage while still allowing large socket buffers to free application resources. With the advent of flash storage devices, which support fast random reads, the secondary storage is a natural candidate for swapping this overflow content.

We used ModNet’s delegation framework to implement a socket buffer swap module that optimizes socket buffering by yanking data from slowly-draining socket buffers and swapping it to the SSD once the socket buffer and the designated per-connection memory are full. Although swapping of excess content to secondary storage increases the throughput when the bottleneck is network, it can degrade the throughput if disk becomes the bottleneck. In order to address this issue, we implement an adaptation mechanism to prevent swapping of new content once the disk load is high. The swap module decides whether or not to swap content by comparing the network bandwidth and the expected disk throughput, which is estimated based on the number of outstanding operations.

It is worth emphasizing here that ModNet's delegation framework makes it very easy to deploy such system-wide policies, and allows prioritizing the scheduling of such performance critical processes. Additionally, since the modules are standalone processes it also makes the process secure by running the module with appropriate privileges to the swap area. Moreover, the inspection mechanism allows us to examine network conditions and decide when to swap. Revocation is used for swapping existing data in case of sudden changes in conditions.

For this experiment, we use a mix of low and high bandwidth clients. The per client bandwidth for low bandwidth clients varies from ∼175Kbps (making the lower end of aggregate bandwidth 0.2Gbps) to ∼4Mbps (making the higher end of aggregate bandwidth around 4.8Gbps). The high bandwidth clients will collectively receive the residual bandwidth of the server's 10 Gbps link. We use 1200 low bandwidth clients and an equal number of high bandwidth clients, which repeatedly request a dynamically generated file of size 800KB.

For generating dynamic content, we use a PHP script that generates 800KB of data and use Apache to serve it. Fig. 8 shows the throughput of native Apache (Native) and Apache with the socket buffer swap module applied to it (Socket Buffer Swap Module). We set the maximum limit on Apache's worker processes to 512, because increasing the limit beyond that reduces its performance.

We see more than 9X improvement in throughput when there are a number of clients with considerably lower bandwidth. As the bandwidths of these low bandwidth clients increase, their request rate also increases, because each request takes less time to finish. Therefore, the throughput starts dropping after a point because of the increased disk load. Note that it still performs better than blocking on the network, until the point where disk throughput becomes the bottleneck. Once the disk throughput becomes the bottleneck, the adaptation mechanism will try to prevent the further offloading of content to SSD; we see a slightly lower performance after this point because the adaptation mechanism is not perfect.

5.6 Deduplicating Socket Buffers

While caching mechanisms and CDNs can be used to handle flash crowds for static content, scalable server instances are required in the case of dynamic content. With the growing complexity of web pages [11], the memory pressure of socket buffers can become a limiting factor for web server scalability. The socket buffer swap module (§5.5) can be used to reduce the memory pressure at the expense of higher disk I/O. In this section we describe the deduplication module, which reduces the memory pressure at the expense of higher CPU utilization.

The deduplication module exploits the fact that responses generated by web servers often contain large amounts of template material, such as in the case of dynamic content [15]. Web servers can avoid the duplicate buffering for static files by using the `sendfile` system call (or its equivalents). However, there is no easy mechanism to avoid this extra memory pressure for web proxies or web servers generating dynamic content. We implemented a deduplication module using ModNet's delegation framework that reduces duplicate buffering for servers. As argued in §5.5, ModNet modules greatly ease the deployment of such "OS-like" services.

We use Rabin fingerprinting to detect duplicate chunks, as in [9, 29], and share a single copy of the duplicated content by using Linux's `vmsplice` system call. Fig. 9 shows the memory usage versus the number of concurrent connections for Nginx (Native) and Nginx with the deduplication (Deduplication) module applied to it. All the connections request a dynamically generated file with the same template. We used the Yahoo homepage as the template and inserted scripts for portions we deemed would vary across downloads by different users. The size of the page was 346KB on an average, and the dynamic portion was less than 300 bytes. As can be seen in the graph, the memory consumption drops by up to 7X for this experiment.

Note that varying the number of connections does not affect the throughput for this experiment because we are network bound throughout the range of this experiment. The average application throughput was close to 930Mbps for with and without deduplication on the 1Gbps link. The average CPU utilizations were 31% and 63% for Native and Deduplication, respectively. However, for the 10Gbps link, the deduplication was CPU bound and was only able to deliver ∼3.2Gbps of throughput.

The relative CPU overhead of deduplication increases as the content generation processing decreases. Therefore, we emulated zero generation time using static content in order to expose the maximum overhead; the average CPU utilizations for Native and Deduplication were 17% and 52% in this case. Thus, the maximum processing overhead of deduplication is around 200%. However,

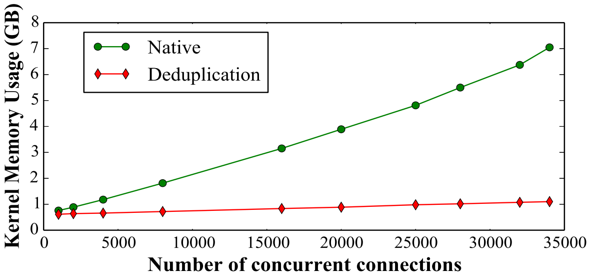


Figure 9: Kernel memory usage for Native Nginx and Nginx with the deduplication module.

the fact that the deduplication module was able to saturate the 1Gbps link with a moderately powerful 4-core server makes this a highly practical solution.

5.7 Video Streaming with Revocation

While the swapping and deduplication modules yank and later re-insert content into the socket buffer, another use of ModNet's revocation mechanism is to change unsent content in the socket buffer. We target adaptive bitrate video, where the client requests video fragments encoded at multiple bitrates. If the client chooses the wrong bitrate or if the network connection abruptly changes, the viewer can experience rebuffering. As discussed in §5.4, ModNet's shared state mechanism can be used for packet-pair based passive bandwidth estimation.

We implement yank support in Mistserver [8], an open source HLS [22] server, so that the server can participate in the adaptive bitrate system. We implement two approaches – in the first, the server monitors bandwidth and truncates any in-progress transfer, leaving any content already in the socket buffer to be sent. In the other, not only does any remaining content get stopped, but any unsent data is also yanked from the socket buffer. Although truncation is not officially supported by the HLS [22] protocol, we have tried our implementation with two popular players, VLC and Quicktime, and both of them were able to play the video without any visible problems. In order to support persistent connections, we use the chunked encoding to produce HTTP responses of variable size. We use "Big Buck Bunny" [1] as the video clip for streaming. The duration of segments and encoding levels were chosen in accordance with HLS best practices [3].

We serve the highest bitrate video stream that can be sustained by the current estimated bandwidth. If the bandwidth drops below the bitrate of the current encoding, we truncate the video segment. While truncating, we can use the `yank` system call to remove the pending socket buffer data at any legal boundary. The server prefixes any pending video fragment from the last segment in each response. The entire segment in a response is encoded at a bitrate that can be sustained by the bandwidth that was recorded at the end of transmission of the last segment.

Fig. 10 shows a plot of segment bitrates vs. time for the requested segments for a synthetic bandwidth variation dataset, using VLC as the client. Researchers have conducted similar studies in the past [10] for other adaptive HTTP-based video streaming protocols. Results are shown for the following three variants: (1) adaptive, which is the default HLS behavior, (2) truncate, which uses server-side adaptation and truncation of segments, and (3) yank, which uses server-side adaptation and employs yank to perform better truncation. The plot clearly demonstrates that the standard adaptive protocol reacts very slowly to steep bandwidth changes, a server-side truncation mechanism allows a relatively faster reaction, and using yank allows us to react almost instantaneously.

Fig. 11 shows the re-buffering durations, where the player has no video segments to play, and the startup times before the video playback begins, for the different bandwidth traces of a real 3G network [25] (see table 2 for details). Our version of VLC starts the playback as soon as it has downloaded one full segment; earlier versions used to start playback after downloading two full segments. Note that we have only shown the results for a small set of representative traces that exhibit some amount of re-buffering.

From this test, we see that simply using server-side adaptation to truncate ongoing segments and change bitrate can yield some improvement over client-side adaptation. However, being able to use ModNet's `modnet_yank` can reduce the rebuffering and startup time by as much as a factor of 2-5 for these traces. At the same time, the normal operation of the server is not impacted, since it can continue to use large socket buffers when the client's bandwidth estimation is correct.

6 Related Work

ModNet can be viewed as a combination of an interposition system and a proxy, although it has more network interaction than either of those systems. In this section, we discuss related systems that have not already been mentioned in the paper.

Various kinds of proxies can provide some of the same kinds of behaviors we have shown in this paper. Fox et. al. [13] propose the use of transformational proxies to perform compression of content according to network bandwidth, screen size and other client's characteristics, but heavily rely on client-side support for bandwidth estimation. Sucu et. al. [30] propose a mechanism for adaptively compressing the network content using bandwidth estimation mechanisms from grid computing research. ModNet's interfaces make our Gzip module im-

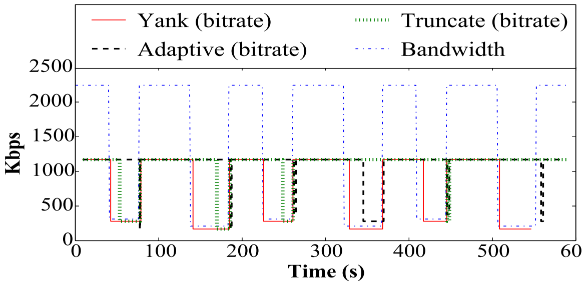


Figure 10: The bitrate of video segments for a synthetic bandwidth trace for all three variants: adaptive, truncate and yank.

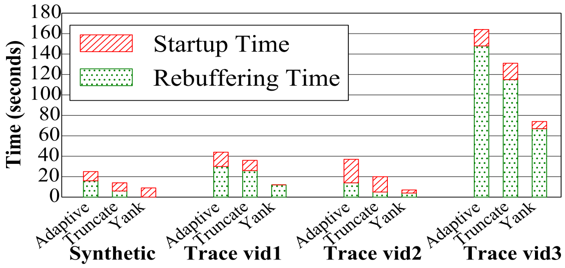


Figure 11: Rebuffering durations and startup times for different video traces using different adaptation strategies. Yanking content dramatically reduces rebuffering times in all environments tested.

plementation more straightforward. Krishnamurthy et. al. [17] propose characterization of clients based on network connectivity for adapting web server responses, which is a server-side response to similar work that was done amongst clients by Seshan et. al. [28]. We believe that ModNet's interface for exposing connection status information makes this process more direct, and can augment client-side estimation as shown in our adaptive video experiment.

Other proxy work has implemented portions of the work in ModNet. Rosu et. al. [26] propose a shared memory abstraction for exposing some socket state information to applications. However, their main intention behind doing so is to implement a fast select/poll mechanism at user level. Connection conditioning [23] also uses a chained series of services. However, their mechanism is specific to web servers, and only handles the request path, not responses. Furthermore, their implementation, which is purely in user space, is considerably different than ours, and demands application changes. Other loopback proxy approaches have much higher performance overheads, as we show in the microbenchmark experiments.

Packet interception mechanisms, such as packet filtering [20, 21] or virtual network devices such as TUN/TAP in Linux, might allow a user space daemon to intercept the packets and modify them (e.g. the Linux libnetfilter_queue [34] mechanism). Since the daemon intercepts individual packets, it is not suitable for connection-oriented processing. Specifically, doing things like multiplexing connections through read/write events or `mmap`-ing sockets to read the connection state will not be possible. Even if some connection tracking mechanism was to be used in conjunction, these would demand extra programmer effort.

Much work in general has taken place on user-level network stacks, with the goal of avoiding the long delay in kernel adoption. Some implementations [12, 31] allow some application-level flexibility, although the development and maintenance efforts may make them unattractive for many domains. The most successful user-level stack is arguably Click [16], which has shown that flexibility can be more desirable than raw speed, and which has shaped some of our design choices. Not all user level approaches have implemented the full stack. Tesla [27] is a framework for transparently implementing session-layer services, such as compression, encryption, etc. Although Tesla is more specialized for session-layer services, ModNet's module framework is more generic.

In comparison to user-level stacks, some work has been done on entirely new protocols to avoid these problems, such as DCCP [18]. This protocol implementation provides a shared packet ring abstraction to allow manipulation of buffered data, which they call *late-data choice*. The problem with this approach, however, has been slow deployment at end hosts, limiting application adoption.

7 Conclusions

We present the design and implementation of ModNet, which increases the flexibility of network stack by introducing a framework for delegating network stack management, inspecting connection progress, and revoking unsent content. We demonstrate a range of modules that allow dynamic control of data generation, socket buffer management, and server behavior, at time scales and granularities not easily achieved with existing interfaces. We believe that the small amount of kernel change introduced by ModNet is palatable, and the additional mechanism is small, general-purpose, and can be easily maintained, raising the chances that ModNet or something like it will have greater deployability than custom protocols or other approaches with higher barriers to adoption.

8 Acknowledgements

We would like to thank our shepherd, Eddie Kohler, as well as the anonymous NSDI reviewers. This research was supported by NSF Award CNS-1217782.

References

[1] Big Buck Bunny: http://www.bigbuckbunny.org/.

[2] Google Chrome data compression proxy: https://developer.chrome.com/multidevice/data-compression.

[3] HLS Best Practices: https://developer.apple.com/library/ios/technotes/tn2224/_index.html.

[4] JPEG Specifications: http://www.w3.org/Graphics/JPEG/itu-t81.pdf.

[5] Libevent: http://libevent.org/.

[6] Linux Advanced Routing & Traffic Control: http://www.lartc.org/manpages/tc.html.

[7] Microsoft smooth streaming:http://www.microsoft.com/silverlight/iis-smooth-streaming/.

[8] The Mistserver wiki: http://wiki.mistserver.org/.

[9] B. Agarwal, A. Akella, A. Anand, A. Balachandran, P. Chitnis, C. Muthukrishnan, R. Ramjee, and G. Varghese. Endre: An end-system redundancy elimination service for enterprises. In *USENIX NSDI*, 2010.

[10] S. Akhshabi, A. C. Begen, and C. Dovrolis. An experimental evaluation of rate-adaptation algorithms in adaptive streaming over HTTP. In *Proceedings of the second annual ACM conference on Multimedia systems*, pages 157–168. ACM, 2011.

[11] M. Butkiewicz, H. V. Madhyastha, and V. Sekar. Understanding website complexity: measurements, metrics, and implications. In *Proceedings of the 2011 ACM SIGCOMM Internet measurement conference (IMC)*, pages 313–328. ACM, 2011.

[12] D. Ely, S. Savage, and D. Wetherall. Alpine: A user-level infrastructure for network protocol development. In *USITS*, volume 1, pages 15–15, 2001.

[13] A. Fox, S. D. Gribble, E. A. Brewer, and E. Amir. Adapting to network and client variability via on-demand dynamic distillation. In *Proceedings of the Seventh International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS VII, pages 160–170, New York, NY, USA, 1996. ACM.

[14] A. Fox, S. D. Gribble, Y. Chawathe, and E. A. Brewer. Adapting to network and client variation using infrastructural proxies: Lessons and perspectives. *Personal Communications, IEEE*, 5(4):10–19, 1998.

[15] D. Gibson, K. Punera, and A. Tomkins. The volume and evolution of web page templates. In *Special Interest Tracks and Posters of the 14th International Conference on World Wide Web (WWW)*, WWW '05, pages 830–839, New York, NY, USA, 2005. ACM.

[16] E. Kohler, R. Morris, B. Chen, J. Jannotti, and M. F. Kaashoek. The Click modular router. *ACM Transactions on Computer Systems (TOCS)*, 18(3):263–297, 2000.

[17] B. Krishnamurthy and C. E. Wills. Improving web performance by client characterization driven server adaptation. In *Proceedings of the 11th international conference on World Wide Web*, pages 305–316. ACM, 2002.

[18] J. Lai and E. Kohler. Efficiency and late data choice in a user-kernel interface for congestion-controlled datagrams. In *SPIE*, volume 5680, pages 136–142, 2005.

[19] K. Lai and M. Baker. Measuring link bandwidths using a deterministic model of packet delay. In *ACM SIGCOMM Computer Communication Review*, volume 30, pages 283–294. ACM, 2000.

[20] S. McCanne and V. Jacobson. The BSD packet filter: A new architecture for user-level packet capture. In *USENIX Winter 1993 Conference*, page 2, 1993.

[21] J. C. Mogul, R. F. Rashid, and M. J. Accetta. The packet filter: An efficient mechanism for user-level network code. In *SOSP*, pages 39–51. ACM, 1987.

[22] R. Pantos. HLS Internet Draft: http://tools.ietf.org/html/draft-pantos-http-live-streaming-12.

[23] K. Park and V. S. Pai. Connection conditioning: Architecture-independent support for simple, robust servers. In *USENIX NSDI*, 2006.

[24] A. Pesterev, J. Strauss, N. Zeldovich, and R. T. Morris. Improving network connection locality on multicore systems. In *Proceedings of the 7th ACM European conference on Computer Systems*, pages 337–350. ACM, 2012.

[25] H. Riiser, P. Vigmostad, C. Griwodz, and P. Halvorsen. Commute path bandwidth traces from 3G networks: analysis and applications. In *Proceedings of the 4th ACM Multimedia Systems Conference*, pages 114–118. ACM, 2013.

[26] M. C. Rosu and D. Rosu. Kernel support for faster web proxies. In *USENIX Annual Technical Conference*, pages 225–238, 2003.

[27] J. Salz, H. Balakrishnan, and A. C. Snoeren. Tesla: A transparent, extensible session-layer architecture for end-to-end network services. In *USENIX Symposium on Internet Technologies and Systems*, 2003.

[28] S. Seshan, M. Stemm, and R. H. Katz. Spand: Shared passive network performance discovery. In *USENIX Symposium on Internet Technologies and Systems*, pages 1–18, 1997.

[29] N. T. Spring and D. Wetherall. A protocol-independent technique for eliminating redundant network traffic. *ACM SIGCOMM Computer Communication Review*, 30(4):87–95, 2000.

[30] S. Sucu and C. Krintz. Ace: A resource-aware adaptive compression environment. In *Information Technology: Coding and Computing [Computers and Communications], 2003. Proceedings. ITCC 2003. International Conference on*, pages 183–188. IEEE, 2003.

[31] C. A. Thekkath, T. D. Nguyen, E. Moy, and E. D. Lazowska. Implementing network protocols at user level. *IEEE/ACM Transactions on Networking (TON)*, 1(5):554–565, 1993.

[32] Z. Wang, A. C. Bovik, H. R. Sheikh, and E. P. Simoncelli. Image quality assessment: from error visibility to structural similarity. *Image Processing, IEEE Transactions on*, 13(4):600–612, 2004.

[33] Z. Wang, Z. Qian, Q. Xu, Z. Mao, and M. Zhang. An untold story of middleboxes in cellular networks. *ACM SIGCOMM Computer Communication Review*, 41(4):374–385, 2011.

[34] H. Welte. The libnetfilter_queue home: http://www.netfilter.org/projects/libnetfilter_queue/index.html.

KLOTSKI: Reprioritizing Web Content to Improve User Experience on Mobile Devices

Michael Butkiewicz*, Daimeng Wang*, Zhe Wu*‡, Harsha V. Madhyastha*‡, and Vyas Sekar†

*UC Riverside †CMU ‡University of Michigan

Abstract— **Despite web access on mobile devices becoming commonplace, users continue to experience poor web performance on these devices. Traditional approaches for improving web performance (e.g., compression, SPDY, faster browsers) face an uphill battle due to the fundamentally conflicting trends in user expectations of lower load times and richer web content. Embracing the reality that page load times will continue to be higher than user tolerance limits for the foreseeable future, we ask: How can we deliver the best possible user experience?**

To this end, we present KLOTSKI, a system that *prioritizes* the content most relevant to a user's preferences. In designing KLOTSKI, we address several challenges in: (1) accounting for inter-resource dependencies on a page; (2) enabling fast selection and load time estimation for the subset of resources to be prioritized; and (3) developing a practical implementation that requires no changes to websites. Across a range of user preference criteria, KLOTSKI can significantly improve the user experience relative to native websites.

1 Introduction

Web access on mobile platforms already constitutes a significant (more than 35%) share of web traffic [28] and is even projected to surpass traditional modes of desktop- and laptop-based access [19, 23]. In parallel, user expectations of web performance on mobile devices are increasing. Industry analysts report that 71% of users expect websites to load as quickly as on their desktops, and 33% of annoyed users are likely to visit competing sites, resulting in lost revenues [30, 10].

To cater to this need for a faster mobile web, there are a range of proposed solutions such as customizing content for mobile devices [18, 7], specialized browsers [21], in-cloud acceleration solutions for executing scripts [2], new protocols [25], and compression/caching solutions [2, 8]. Despite these efforts, user experience on mobile devices is still woefully short of user expectations. Industry reports show that the median web page takes almost 11 seconds to load over mobile networks even on state-of-art devices [1]; this is the case even for top mobile-optimized retail websites [15]. In fact, several recent studies show that the benefits from the aforementioned optimizations are marginal [37, 59, 3], and they may even hurt performance [56].

Our thesis is that the increasing complexity of web page content [12, 5, 33] and decreasing user tolerance will outpace the benefits from such incremental performance enhancements, at least for the foreseeable future. For instance, though RTTs on mobile networks halved between 2004 and 2009 [54], the average number of resources on a web page tripled during the same period [5]. Therefore, rather than blindly try to improve performance like prior approaches, we argue that we need to improve the user experience even if load times will be high.

Our high-level idea is to *dynamically reprioritize* web content so that the resources on a page that are critical to the user experience get delivered sooner. For instance, user studies show a typical tolerance limit of 3–5 seconds [39, 32, 48]. Thus, our goal is to deliver as many high utility resources as possible within this time. Our user studies, however, suggest that the content considered high utility significantly varies across users. Therefore, point solutions that optimize for a single notion of user utility, e.g., by statically rewriting web pages or by dynamically prioritizing above-the-fold objects [14, 35] will not suffice. Instead, we want to develop a general solution that can handle arbitrary user preferences.

However, there are three key challenges in making this approach practical:

- **Inferring resource dependencies:** Scheduling the resources on a web page requires a detailed understanding of the loading dependencies between them. This is especially challenging for dynamically generated web content, which is increasingly common.
- **Fast scheduling logic:** We need a fast (tens of ms) scheduling algorithm that can generate near-optimal schedules for arbitrary user utility functions. The challenge is that this scheduling problem is NP-hard and is inefficient to solve using off-the-shelf solvers.
- **Estimating load times:** Predicting the load time for a given web page is hard due to the complex manner in which browsers parallelize the loading of resources on a web page. Our problem is much worse—we need to estimate the load times for arbitrary loading schedules for subsets of web resources. Furthermore, we need to be able to do so across heterogeneous device and network conditions.

In this paper, we present the design and implementation of KLOTSKI, a practical dynamic reprioritization layer that delivers better user experience. Conceptually, KLOTSKI consists of two parts: a back-end measurement engine and a front-end proxy. The back-end uses offline measurements to capture key invariant characteristics of a web page, while the front-end uses these characteristics along with user preferences and client conditions to prioritize high-utility content. In tackling the above challenges, KLOTSKI's design makes three contributions:

- Though the specific URLs on a page vary across loads, we develop techniques to merge multiple loads of a page to extract the page's invariant dependency structure and capture how resource URLs vary across loads.
- We design a fast and near-optimal greedy algorithm to identify the set of resources to prioritize.
- We create an efficient load time estimator, based on the insight that the key bottleneck is the link between the client and the KLOTSKI front-end. Thus, we can effectively simulate this interaction to estimate load times.

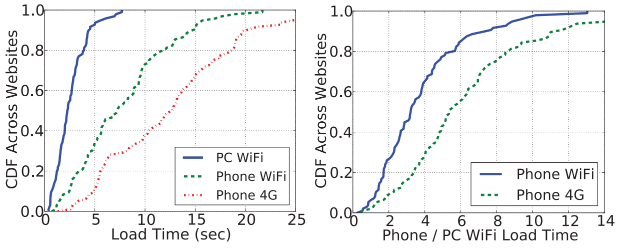
We implement KLOTSKI as an integrated proxy-browser architecture [21, 2] that improves user experience on legacy devices and web pages by using standard web protocols to implement our reprioritization scheme. Using a range of measurements, system benchmarks, and across a variety of user utility functions, we demonstrate that: (1) on the median web page, KLOTSKI increases the fraction of high utility resources delivered within 2 seconds from 25% to roughly 60%; (2) our dependency representations are robust to flux in page content and typically only need to be updated once every 4 hours; and (3) our load time estimates achieve near-ideal accuracy.

Looking beyond our specific design and implementation, we believe that the principles and techniques in KLOTSKI can be more broadly adopted and are well aligned with emerging web standards [6, 13, 25]. Moreover, while our focus here is on mobile web access, we show that KLOTSKI can also improve traditional desktop browsing as well.

2 Motivation

We begin by confirming that: 1) web performance on mobile devices is still below expectations, and 2) these performance issues exist even with popular optimizations. We also argue that these issues stem from the growing complexity of web content and that this growth is outpacing improvements in network performance.

Web performance on mobile devices: The growing adoption of the mobile web has been accompanied by a corresponding decrease in user tolerance—users today expect performance comparable to their desktops on their phones [30]. To understand the state of mobile web performance, we compared the page load times[1] of the landing pages of the top 200 websites (as ranked by Alexa) under three scenarios: 1) on a HTC Sensation smartphone using a 4G connection, 2) on the same phone using WiFi, and 3) on a desktop connected to the same WiFi network. For each web page, we run these three scenarios simultaneously to avoid biases due to content variability across loads. For each page, we report the median load time across 5 loads.



(a) (b)

Figure 1: ***Load time comparison for top 200 websites.***

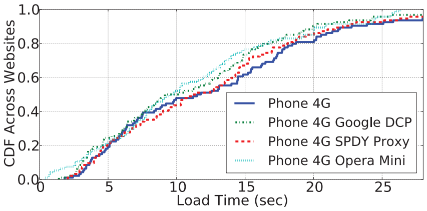


Figure 2: ***Comparison of page load times with various well-known performance optimization techniques.***

Figure 1(a) shows the CDF of load times for the three scenarios, and Figure 1(b) shows the *normalized* load times on the smartphone w.r.t. the desktop case. We see that the mobile load times are significantly worse, e.g., the median webpage is **5**× worse on 4G and **3**× worse even on WiFi. The tail performance is particularly bad, with the 80^{th} percentile $\geq$ 10s on both 4G and WiFi.

Limitations of performance optimizations: We study three prominent classes of web optimizations used today: split-browsers such as Opera Mini [21], Google SPDY [25], and recommended compression strategies. For the latter two cases, we relay page loads through a SPDY-enabled NodeJS proxy and through Google's data compression proxy (DCP) [8], respectively. We use the HTC Sensation smartphone with a 4G connection for these measurements.

Initially, we loaded the top 200 websites using these performance optimizations. However, we saw no improvement in load times (not shown). To see if these optimizations can potentially help *other* websites, we pick the landing pages of 100 websites chosen at random from the top 2000 websites and compare the load times with and without these optimizations in Figure 2. While the optimizations help improve load times on some web pages, we see that they increase load times on other web pages, thus resulting in little change in the overall distribution; load times remain considerably higher than the 5-second tolerance threshold typically observed in usability studies [30]. These observations are consistent with other recent studies [59, 37, 56].

Complexity vs. Performance: A key reason why these protocol-, network-, and browser-level optimizations are largely ineffective is because web pages have

[1] We measure page load time by the time between when a page load was initiated and when the browser's `onLoad` event was fired.

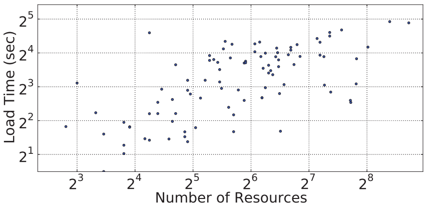


Figure 3: ***Page load time when using Google's compression proxy vs. the number of resources loaded on the page.***

become *highly complex* [12, 5, 33]. For instance, the number of resources included on a web page is a key factor that impacts load times [58, 33]. Indeed, even in our measurements, Figure 3 shows that the load time for every web page using Google's compression proxy shows a strong correlation with the number of resources on the page; i.e., even with the optimizations, the number of resources continues to be a dominant factor.

Furthermore, past trends indicate that increase in page complexity tends to match or even outpace improvements in network performance. For example, prior studies show that the average number of resources on a web page tripled from 2004 to 2009 [5], while RTTs on mobile networks only halved over the same period [54].

Takeaways: In summary, we see that web page loads take a significant amount of time on mobile devices, and that common optimizations offer limited improvements for complex pages. Given that page complexity is likely to grow at the same or faster rate than improvements in network performance, we need to rethink current approaches to improve the mobile web experience.

3 System overview

Embracing the reality that load times will be high despite performance optimizations, we argue that rather than purely focusing on improving performance, we should be asking a different question:

How can we deliver good user experience under the assumption that page load times will be high?

In this section, we start with the intuition underlying our approach and discuss practical challenges associated with realizing this goal. Then, we present an overview of the KLOTSKI system to address these challenges.

3.1 Approach and Challenges

Our high-level approach is to ensure that resources that the user considers important are *delivered sooner*. Note that we do not block or filter any content, so as to not risk rendering websites unusable. Based on studies showing that users have a natural frustration or tolerance limit of a few seconds [39, 32, 48], our goal is to deliver as many high utility URLs on the page as possible within a (customizable) tolerance threshold of M seconds.

To see how this idea works, consider a hypothetical "oracle" proxy server whose input is the set of all URLs $O = \{o_i\}$ on a web page. Each o_i has an associated load time t_i and a user-perceived utility $Util_i$. The oracle picks a subset of URLs $O' \subseteq O$ that can be loaded within the time limit M such that this subset maximizes the total utility $\sum_{o_i \in O'} Util_i$. The proxy will then prioritize the delivery of these selected URLs.

Using this abstract problem formulation, we highlight several challenges:

- *Page dependencies and content churn:* First, this subset selection view ignores inter-resource dependencies, e.g., when a page downloads an image as a result of executing a script on the client, the script is a natural *parent* of the image. To prioritize a high-utility URL o_i, we must also prioritize *all* of o_i's ancestors. Second, because dynamically generated content is common on today's web pages, we may not even know the set O of URLs before loading the page.
- *Computation time:* Selecting the subset O' that maximizes utility is NP-hard even ignoring dependencies, and adding dependencies makes the optimization more complex. Since the number of URLs fetched on a typical web page is large ($\approx$ 100 URLs [4]), it is infeasible to exhaustively evaluate all possible subsets. Note that running this step offline does not help as it cannot accommodate diversity across user preferences and operating conditions (e.g., 3G vs. LTE).
- *Estimating load times:* Any algorithm for selecting URLs to prioritize will need to estimate the load time for any subset of URLs, to check that it is $\leq M$. This estimation has to be reasonably accurate; underestimation will result in some of the selected high utility URLs failing to load within the user's tolerance threshold, whereas over-estimating and choosing additional high utility URLs to load if all the selected URLs load well within the time limit M may lead to suboptimal solutions. Unfortunately, predicting the load time for a given subset of URLs is non-trivial. In addition to the dependencies described above, it is hard to model how browsers parallelize requests, parse HTML/CSS files, and execute scripts.
- *Deployment considerations:* Requiring custom features from clients or explicit support from providers reduces the likelihood of deployment and/or restricts the benefits to a small subset of users and providers. Thus, we have a practical constraint—the prioritization strategy should be realizable even with commodity clients and legacy websites.

3.2 KLOTSKI Architecture

To tackle the above challenges, we develop the KLOTSKI system shown in Figure 4. We envision KLOTSKI as a cloud-based service for mobile web acceleration. There are many players in the mobile ecosystem who have natural incentives to deploy such a service, including browser vendors (e.g., Opera Mini), device vendors (e.g.,

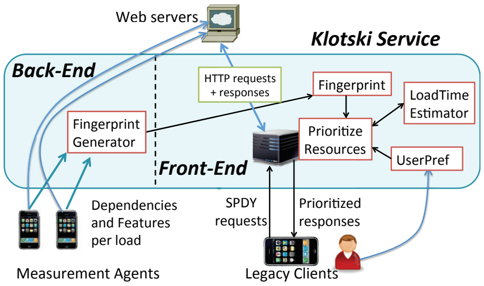


Figure 4: ***Overview of* KLOTSKI*'s architecture.***

Kindle Fire's Silk), cellular providers (offering KLOTSKI as a value-added service), and third-party content delivery platforms (e.g., Akamai). While KLOTSKI requires no changes to client devices or legacy webservers and makes minimal assumptions about their software capabilities, it can also incorporate other optimizations (e.g., caching, compression) that they offer.

We assume there is some process for KLOTSKI users to specify their preferences; we discuss some potential approaches in Section 9. Our focus in this paper is on building the platform for content reprioritization, and we defer the task of learning user preferences to future work.

The KLOTSKI **back-end** is responsible for capturing page dependencies and dynamics via offline measurements using *measurement agents*.[2] For every page fetched, the agents record and report key properties such as the dependencies between resources fetched, the size (in bytes) of every resource, the page load *waterfall*, and every resource's position on the rendered display. The back-end aggregates different measurements of the same page (across devices and over time) to generate a compact *fingerprint* f_w per web page w. At a high-level, f_w is a DAG, where each node i is associated with a *URL pattern* p_i. (The role of this URL pattern will become clearer below.) In this paper, we focus specifically on the fingerprint generation algorithm (§4) and do not address issues such as coordinating measurements across agents.

The KLOTSKI **front-end** is an enhanced web proxy that *prioritizes* URLs that the user considers important. It uses legacy HTTP to communicate with webservers, and communicates with clients using SPDY, which is now supported by popular web browsers [26]. When a request for a page w from user u arrives (i.e., the GET for `index.html`), the front-end uses f_w, the user's preferences, and a load time estimator (§6) to compute the set of resources that should be prioritized (§5).

The front-end can preemptively *push* static resources that need to be prioritized. For other selected resources that are dynamic, however, it cannot know the URLs in the current load until the page load actually executes. Thus, when a new GET request from the client arrives, the front-end *matches* the URL requested against the URL patterns for the selected resources. If a match is found, the front-end prioritizes the delivery of the content for these URLs over other active requests.

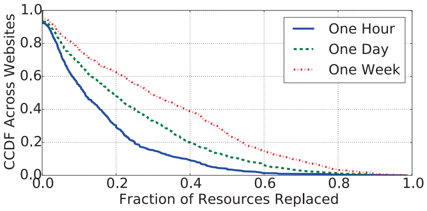


Figure 5: ***Fraction of replaced URLs, comparing loads an hour apart, a day apart, and a week apart.***

4 Page fingerprint generation

Next, we describe how the KLOTSKI back-end generates web page fingerprints. It takes as input multiple loads of a given webpage w as input, and generates the fingerprint f_w that captures parent-children dependencies across resources on w as well as high-level URL patterns describing each resource on the page.

4.1 High-level approach

Prior works such as WebProphet [42] and WProf [58] infer dependencies across URLs *for a single load* of a web page. Unfortunately, this single-load dependency graph cannot be used as our f_w because the URLs on a page change frequently. Figure 5 shows a measurement of the URL churn for 500 web pages. We see that at least 20% of URLs are replaced for $\geq$ 30% of web pages over the course of an hour and for $\geq$ 60% of web pages over a week. Due to this flux in content, the dependencies inferred from a prior load of w may cause us to incorrectly prioritize URLs that are no longer on the page or fail to prioritize the new parent of a high utility URL.

Now, even though the set of URLs changes across page loads, there appears to be an intrinsic *dependency structure* for every web page that remains relatively static. Suppose we construct an abstract DAG from every load of a web page, with an edge from every URL o to its parent p. Then, for 90% of the same 500 pages considered above, changes in this DAG, captured by tree edit distance, are less than 20% *even after a week* (not shown). That is, most pages appear to have a stable set of inter-resource dependencies, with only the URL corresponding to every resource changing across loads.

Based on this insight, we generate the fingerprint f_w as follows. We take multiple measurements of w over a recent interval of Δ hours. From this set of measurements, $Loads_{w,\Delta}$, we identify a *reference load* $RefLoad_w$. While there are many possible choices, we find using the load with the median page load time within $Loads_{w,\Delta}$ works well. Given that the dependency structure is quite stable, we use the dependency DAG for

[2] The measurement agents can be KLOTSKI's clients that occasionally run unoptimized loads, or the KLOTSKI provider can use dedicated measurement clients (e.g., [16]).

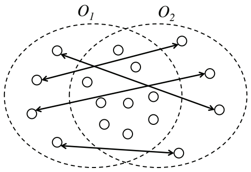


Figure 6: ***Given O_1 and O_2 fetched in two loads of the same web page, we map URLs in $(O_1 - O_2)$ to ones in $(O_2 - O_1)$.***

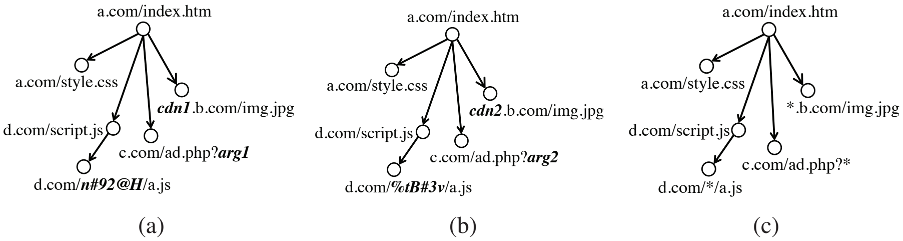


Figure 7: ***Illustrative example showing dependency structures (a) and (b) being merged into aggregated dependency structure (c).***

$RefLoad_w$; in Section 7, we describe how we obtain this dependency DAG in our implementation of KLOTSKI.

However, one challenge remains. To ensure that this DAG is reusable across loads, we need to annotate every node in the DAG with a *URL pattern* that captures the variations of the URLs for this node across loads. We do this as follows. We initialize the page's dependency structure D as the DAG between URLs fetched during the reference load $RefLoad_w$. We then iteratively update D by reconciling the differences between D and other loads in $Loads_{w,\Delta}$. First, for every $o \in RefO$ (the set of URLs in $RefLoad_w$) that is absent in some other (non-reference) load O, we identify the URL $o' \in O$ that replaced o. Then, for every such matched URL, we update the URL pattern at the node in D with a regular expression that matches both the previous URL annotation and the URL for o'.

Thus, we have two natural subtasks: (1) Given two different loads $Load_1$ and $Load_2$, we need to map URLs that have replaced each other; and (2) We need to capture the variations in a resource's URL across loads to generate robust URL patterns. We describe these next.

4.2 Identifying replacements in URLs

Let O_1 and O_2 be the sets of URLs fetched in two different loads within $Loads_{w,\Delta}$. While some URLs are present in both loads, some appear only in one. Our goal here is to establish a bijection between URLs fetched only in the first load ($O_1 - O_2$) and those fetched only in the second load ($O_2 - O_1$), as shown in Figure 6.

```
<span class="yt-thumb-default">
<span class="yt-thumb-clip">
<img aria-hidden="true" alt="" width="175"
src="//i1.ytimg.com/vi_webp/k2waw0wZ7VA/mqdefault.webp">
<span class="vertical-align">
</span></span></span></span>
```

(a) Load 1

```
<span class="yt-thumb-default">
<span class="yt-thumb-clip">
<img aria-hidden="true" alt="" width="175"
src=" //i1.ytimg.com/vi/GSjA3voJydk/mqdefault.jpg">
<span class="vertical-align">
</span></span></span></span>
```

(b) Load 2

Figure 8: ***Example snippets from two loads of `youtube.com` that illustrate the utility of local similarity based matching. URLs that replace each other are shown in bold.***

To this end, we rely on three key building blocks:

- **Identical parent, lone replaced child:** We identify the parent(s) for each URL in O_1 and O_2. Then, we consider every URL p that appears in both loads and has children in both loads. Now, if p has only one unmatched child o in O_1 and only one unmatched child o' in O_2, i.e., p's remaining children appear in both loads[3], then we consider o' to have *replaced* o.
- **Similar surrounding text:** In practice, a single parent resource p may have several children replaced across loads; e.g., a script may fetch different URLs across executions, or the URLs referenced in a page's HTML may be rewritten across loads. In such cases, we need to identify the mapping between an URL p's children in $(O_1 - O_2)$ and its children in $(O_2 - O_1)$.
  Here, we observe that the relative position of these children in their parent's source code is likely to be similar. Figure 8 shows an example snippet from the main HTML on `youtube.com`, which fetches different images across loads. As we can see, even as the HTML gets rewritten to fetch a different image, the text within the code surrounding the reference to either image is almost identical.
  We use this observation to identify URL replacements as follows. For every URL, KLOTSKI's measurement agents log the location within its parent's source code where it is referenced. Then, for every pair (o, o') that have the same parent p, we compute a *local text similarity score* between the portions of p that reference o in the first load and o' in the second load. We compute this similarity score as the fraction of common content[4] across (1) 500 characters of text on either side of the URL's reference, and (2) the lines above and below the URL's reference. We iterate over (o, o') pairs in decreasing order of this score, and declare o'_i as having replaced o_i if either URL has not already been paired up and if the score is greater than a threshold.
- **Similar position on display:** Local similarity may fail to identify all URL replacements as not all URLs are directly referenced in the page's source code (e.g., algorithmically generated URLs). Hence, we also iden-

[3] Or equivalently, all the other children have already been matched.

[4] We apply Ratcliff and Metzener's pattern matching algorithm [52], which returns a value in $[0, 1]$ for the similarity between two strings.

tify URL replacements based on the similarity in their position on the display when the page is rendered. Again, KLOTSKI's measurement agents log the coordinates of the top-left and bottom-right corner of the visible content of every URL (details in §7). We then declare o' in one load as having replaced o in another load if the sum of the absolute differences between the coordinates of their corners is less than 5 pixels.

Putting things together: We combine the above building blocks as follows. First, we map URLs that have an identical parent and are the only child of their parent that changes across loads. We apply this technique first since it rarely yields false positives. Then, we identify mappings based on the local similarity scores, following which we leverage similarity in screen positions. After these steps match many URLs in O_1 and O_2, there may now be new URLs that share a parent and are the only unmatched child of their parent. Thus, we apply the first step again to further match URLs. At this point, there remain no more URL pairs that are the sole replaced children of their common parent and all URLs that can be matched based on similarity in surrounding text or screen position have already been matched.

4.3 Generating URL patterns

Now that we are able to identify how URLs fetched on a web page are replaced across loads, we next discuss how KLOTSKI generates the URL patterns.

While one could use complex algorithms to merge URLs into regular expressions (e.g., [61]), our empirical analysis of thousands of websites shows that over 90% of URL replacements fall in one of three categories:

- **URL argument changes:** When URL o in one load is replaced by o' in another, they often differ only in the associated arguments, i.e., the part of the URL following the '?' delimiter. This is common in advertisements, as the argument is dynamically rewritten to select the ad shown. For example, `c.com/ad.php?arg1` in Figure 7(a) is replaced by `c.com/ad.php?arg2` in Figure 7(b). In such cases, we merge URLs into a regular expression that preserves the common prefix and indicates that any argument is acceptable; `c.com/ad.php?*` in our example.
- **Single token in the URL changes:** Second, when URLs o and o' are split into tokens using '/' as the delimiter, they often differ only in one token. This happens when an image on a page is replaced by another image with the same path name, or when an URL includes a hash value that is randomly generated on every load, e.g., in Figures 7(a) and 7(b), `d.com/%tB#3v/a.js` replaces `d.com/n#92@H/a.js`. Here, the merged URL pattern we create is the URL for o, but the token that differs from o''s URL is replaced with a wildcard; `d.com/*/a.js` in our example.

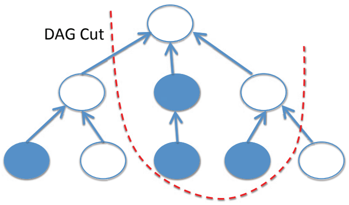


Figure 9: ***Choosing a dependency-compliant subset of resources that maximizes utility within load time budget. Each node represents a resource; shaded nodes have high utility.***

- **Resources fetched from CDNs:** Last, we account for content served via CDNs. For such URLs, the hostname portion of the URL changes across loads only in the first token, when the hostname is split into tokens based on '.'. The regular expression that we use replaces only the portion of the hostname that changes with a wildcard, e.g., in Figure 7, the regular expression `*.b.com/img.jpg` captures `cdn1.b.com/img.jpg` replacing `cdn2.b.com/img.jpg`.

One concern is that these merging techniques may become too generic (i.e., too many wildcards), producing many false matches at the front-end. We show in §8 that, with a suitable choice of Δ to refresh the DAG, this is unlikely to occur.

5 Optimizing page loads

When a client loads a web page w via the KLOTSKI front-end, the front-end does two things. First, it selects the subset of resources on the page that it should prioritize. Thereafter, as the client executes the page load, the front-end alters the sequence in which the page's content is delivered to the client, in order to prioritize the delivery of the selected subset of resources. Next, we discuss how the KLOTSKI front-end performs these tasks.

5.1 Selecting resources to prioritize

Recall from §3.2 that the KLOTSKI front-end begins selecting the subset of resources to prioritize on a page w once it receives the request for w's main HTML.

The front-end's resource selection for w uses the *previously computed* fingerprint f_w that characterizes the dependencies and features of resources on w. Using f_w in combination with the user's preferences, the front-end computes per-resource utilities and constructs an annotated DAG where every node corresponds to a resource on w and is annotated with that resource's utility.

As shown in Figure 9, our goal is to select a suitable *DAG-cut* in this structure, i.e., a cut that also satisfies the dependency constraints. Formally, given a page's dependency structure D and a time budget M for user perceived load time, we want to select the optimal cut C^* that can be loaded within time M and maximizes the expected utility. Now, selecting the optimal cut is NP-hard and it is inefficient to solve using off-the-shelf solvers.[5]

It is clear that we need a fast algorithm for resource subset selection because it is on the critical path for loading web pages—if the selection itself takes too long, it defeats our goal of optimizing the user experience. Hence, we heuristically adapt the greedy heuristic for the weighted knapsack problem as follows.

We associate every resource o_i in the page, whose utility is $Util_i$, with an initial cost C_i equal to the sum of its size and its ancestors' sizes in D. Then, in every round of the greedy algorithm, we iterate through all the unselected resources in the descending order of $\frac{Util_i}{C_i}$. When considering a particular resource, we estimate the time (using the technique in §6) that will be required to load the selected DAG-cut if this resource and all of its ancestors were added to the cut. If this time estimate is within the budget M, then we add this resource and all of its ancestors to the selected DAG-cut; else, we move to the next resource. Every time we add a resource and its ancestors to the DAG cut, we update the cost C_i associated with every unselected resource o_i as the sum of its size and the sizes of all of its ancestors that are not yet in the DAG cut. We repeat these steps until no more resources can be accommodated within the budget M (or all resources have been selected).

5.2 Prioritizing selected resources

Having selected the resources to prioritize, there are two practical issues that remain. First, the front-end does not have the actual content for these resources; f_w only captures dependencies, sizes, and position on the screen. Second, the URLs for many of the resources will only be determined after the client parses HTML/CSS files and executes scripts; the KLOTSKI front-end does not parse or execute the content that it serves.

Given these constraints, the front-end prioritizes transmission of the selected resources to the client in two ways. First, for every static resource (i.e., a resource whose node in the page's f_w is represented with a URL pattern without wildcards), the front-end pre-emptively requests the resource from the corresponding web server and *pushes* the resource's content to the client without waiting for the client to request it. However, the front-end cannot do this for any resource whose URL pattern is not static, as the front-end does not know which of the various URLs that match the URL pattern will be fetched in this particular load. Hence, the front-end matches every URL requested by the client against the URL patterns corresponding to the selected resources, and it prioritizes the delivery to the client of URLs that find a match over those that do not. We describe how we implement these optimizations via `SPDY` features in §7.

6 Load time estimation

As discussed in the previous section, our greedy algorithm needs a load time estimator to check if a candidate subset of resources can be delivered within the load time limit M. In this section, we begin by discussing why some natural strawman solutions fail to provide accurate load time estimation, and then present our approach.

Strawman solutions: One might consider modeling the load time for a subset of resources as some function of key features such as the number of resources, the total number of bytes fetched, or the number of servers/domains contacted. Unfortunately, due to the inter-resource dependencies and the complex (hidden) ways in which browsers issue requests (e.g., interleaving HTML/CSS parsing and script execution vs. actual downloads), these seemingly natural features are poorly correlated with the effective load time. Alternatively, to incorporate the dependencies, we could try to extend the resource loading *waterfall* (i.e., the sequence in which URLs are fetched and the associated timings) from the reference load $RefLoad_w$. However, this approach also has two key shortcomings: (1) since we are explicitly changing the sequence of requests, the original waterfall is no longer accurate, and (2) it is fragile due to the diversity in load times across clients and network conditions.

Our approach: To account for the dependencies and accurately estimate the load time for a given subset of resources, we need to estimate four key timing variables for each URL o_i: (a) $ClientStart_i$, when the client requests o_i; (b) $ProxyStart_i$, when the front-end starts delivering o_i to the client; (c) $ClientReady_i$, when the client can begin to render or use o_i; and (d) $ProxyFin_i$, when the front-end finishes delivering o_i.[6] Together, this gives us all the information we need to model the complete page download process for a given subset of resources.

Intuitively, if the link between the client and the front-end is the only bottleneck and the bandwidth is shared equally across current downloads [46], then we can use a lightweight fluid-model simulation of the client-frontend interaction. Given this assumption, we use a simple analytical model to estimate the values of the four variables as described below. We explain this with the example in Figure 10, where we have 5 URLs with the DAG D shown and everything except o_5 is selected to be prioritized. For clarity of presentation, we describe the case when each o has only one parent.

1. $ClientStart_i$: This depends on the finish time of o_i's parent as well as delays for the client to process the parent; e.g., in Figure 10, o_3 is requested

[5] We can formally prove via a reduction from the weighted knapsack problem, but do not present it for brevity.

[6] All times are specified in terms of the client clock.

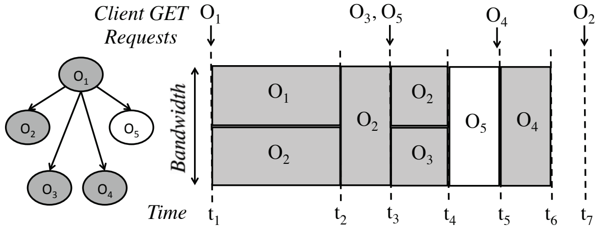


Figure 10: ***Illustrative execution of load time estimator. Shaded resources are high utility resources selected for prioritization. Times shown are for events fired in the simulator.***

some time after the completion of o_1. Specifically, $\mathit{ClientStart}_i = \mathit{ClientReady}_{p_i} + \mathit{Gap}_i$, where p_i is the parent. Gap_i is the processing delay between the parent and the child; we capture these parent-child gaps from KLOTSKI's measurements, store these in f_w, and replay the gaps with a simple linear extrapolation to account for CPU differences between the measurement agent and the current client.

2. $\mathit{ClientReady}_i$: In the simplest case, we can simply set $\mathit{ClientReady}_i = \mathit{ProxyFin}_i$; i.e., when the front-end has finished sending the object. However, there is a subtle issue if the front-end had decided to push URL o_i. In particular, the client may not have processed o_i's parent when the front-end completes delivering o_i. This means that the client will start consuming an URL only when it is ready to issue the request for that URL. Thus, we modify the above expression to $\mathit{ClientReady}_i = \max(\mathit{ProxyFin}_i, \mathit{ClientStart}_i)$. In our example, o_2 finishes downloading at $\mathit{ProxyFin}_i = t_4$, but the client finishes processing the parent to issue the request for o_2 at $\mathit{ClientStart}_i = t_7$.

3. $\mathit{ProxyStart}_i$: The time at which the front-end can start delivering o_i depends on two scenarios. If o_i was chosen to be pushed (see §5.2) , then it can start immediately. Otherwise, the front-end needs to wait until the request arrives (e.g., for dynamically generated URLs). If $\mathit{Latency}$ is the latency between the client and the front-end, we have:

$$\mathit{ProxyStart}_i = \begin{cases} 0, \text{if } o_i \text{ is } \mathit{pushed} \\ \mathit{ClientStart}_i + \mathit{Latency}, \text{otherwise} \end{cases}$$

In our example, the front-end has to wait until the dynamically generated URL o_4 has been requested before starting to deliver it.

4. $\mathit{ProxyFin}_i$: Finally, to compute the time for the front-end to finish delivering an URL, we model the front-end as a priority but work-conserving scheduler with fair sharing. That is, if there are no high-priority URLs to be scheduled, then the front-end will chose some available low priority URL; e.g., in $[t_4, t_5]$, there were no high priority URLs to schedule as o_4 has not yet been requested, so the front-end tries to deliver the low priority URL o_5, but after o_4 is ready, it preempts o_5. Moreover, the bandwidth between the client and the front-end is equally shared across concurrently delivered URLs, e.g., in intervals $[t_1, t_2]$ and $[t_3, t_4]$.

Together, this simple case-by-case analysis provides the necessary information to model the complete page download process for a given subset of resources. As we will see later, our assumptions on the bottleneck link and fair sharing holds reasonably well in practice and this model provides accurate load time estimations.

7 Implementation

Measurement agent: We implement Android-based measurement agents that load web pages in the Chrome browser. We use Chrome's Remote Debugging Protocol to extract the inter-URL dependencies in any particular page load. For every URL fetched, this gives us the mime-type, size, parent, and the position within that parent's source code where this URL is referenced. In addition, when the `onLoad` event in the browser fires, we inject a Javascript into the web page. This script traverses the DOM tree constructed by the browser while loading the page and dumps several pieces of information contained within the node for every resource, e.g., whether it is visible, and if so, its coordinates on screen.

Front-end: We implement the KLOTSKI front-end by modifying the NodeJS proxy with the SPDY module enabled [27]. Our front-end uses SPDY to communicate with clients and HTTP(S) to communicate with webservers. For any resource delivered by the proxy to a client, it maps the resource to one of SPDY's 7 priority levels as follows: a web page's main HTML is mapped to priority 0, pushed resources have priority 1, resources that are dynamically prioritized (by matching their URLs against regular expressions in the web page's fingerprint) are assigned priority 2, and all other resources are spread across lower priority levels in keeping with the order in which the NodeJS proxy assigns priorities by default.

In addition, we require one modification to typical client-side browser configurations in order for them to be compatible with the KLOTSKI front-end. By default, browsers accept resources delivered using SPDY PUSH only if the domain in the resource's URL is the same as the one from which the page is being loaded [25]. We select the configuration option in Chrome for Android which makes it accept pushed resources from any domain. However, since Chrome accepts a HTTPS resource via SPDY PUSH only if it is pushed by the domain hosting it, we consider all such resources only for dynamic prioritization.

8 Evaluation

Our evaluation of KLOTSKI comprises two parts. First, we showcase the improvements in user experience enabled by KLOTSKI across a range of scenarios. Then,

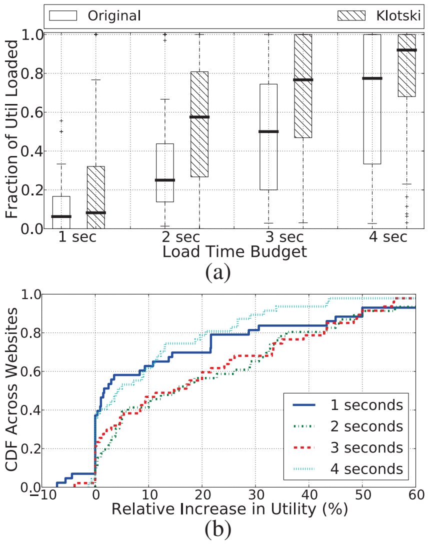


Figure 11: ***Comparison of fraction of high utility resources loaded within load time budget: (a) Box and whisker plots showing spread across websites, and (b) CDF across websites of difference between* KLOTSKI *and the original website.***

we evaluate each of KLOTSKI's components in isolation. We begin with a description of our evaluation setup.

8.1 Evaluation setup

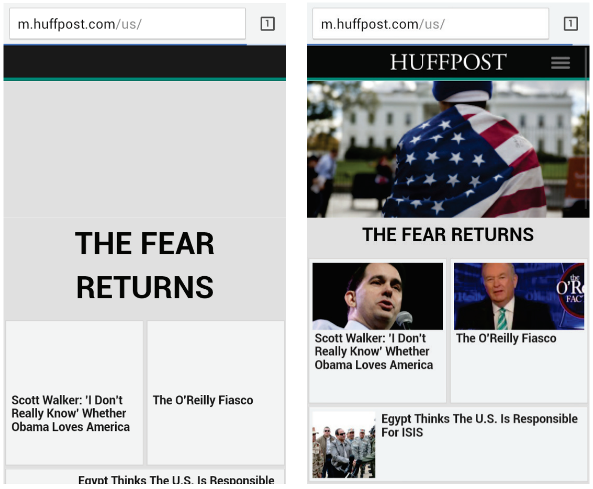
All experiments were conducted using a HTC Sensation smartphone, running Android 4.0.3, as the client. This client connected to a WiFi hotspot exported by a Mac Mini, which in turn obtained its Internet connectivity via a T-Mobile 4G dongle. We use this setup, rather than the phone directly accessing a 4G network, so as to log a pcap of the network transfers during page loads.

For most of our experiments, we use the landing pages of 50 websites chosen at random from Alexa's top 200 websites. We load the full version of these web pages using Google Chrome version 34.0.1847.116 for Android. We host the KLOTSKI front-end in a small instance VM in Amazon EC2's US West region.

8.2 Improvement in user experience

We evaluate the improvement in user experience enabled by KLOTSKI, compared to page loads that go through an unmodified proxy, in a variety of client/network settings and across a range of user preferences; note that we see little difference in load times when our client directly downloads page content from webservers and when it does so via a vanilla web proxy. In all cases, though resources not visible to the user (e.g., CSS and Javascripts) have a utility score of 0, KLOTSKI may choose to prioritize such resources if doing so is necessary in order to prioritize a high utility resource, due to dependencies.

Prioritizing above-the-fold content: First, we consider all resources on a page that appear "above-the-fold" (i.e., resources that are visible without the user having to


m.huffpost.com/us/

THE FEAR RETURNS

Scott Walker: 'I Don't Really Know' Whether Obama Loves America

The O'Reilly Fiasco

Egypt Thinks The U.S. Is Responsible

m.huffpost.com/us/

HUFFPOST

THE FEAR RETURNS

Scott Walker: 'I Don't Really Know' Whether Obama Loves America

The O'Reilly Fiasco

Egypt Thinks The U.S. Is Responsible For ISIS

(a) Original page load (b) Page load with KLOTSKI

Figure 12: ***Screenshots comparing loads of an example site (`http://huffpost.com`), 3 seconds into the page load, without and with* KLOTSKI.**

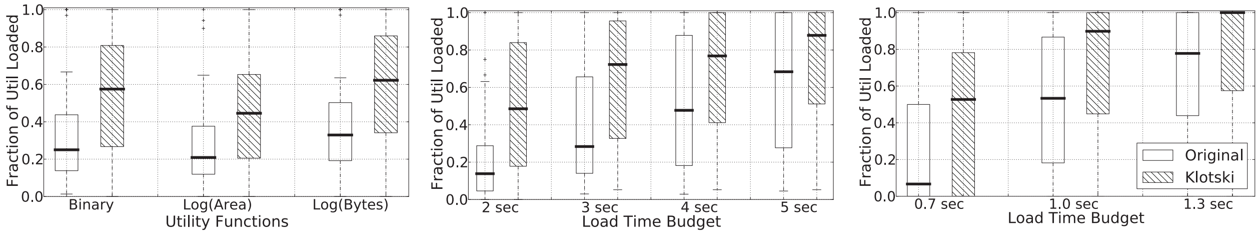
scroll) as high utility. We assign a utility score of 1 for every high utility object and a score of 0 for all others.

We then load every web page on our smartphone client first without any optimization, and then via the KLOTSKI front-end. In either case, we log the sequence in which resources were received at the client and later identify the high utility resources delivered within the load time budget. We ran this experiment varying the load time budget value between 1 and 4 seconds; prior studies suggest most users have a tolerance of at most 5 seconds [30].

For each load time budget value, Figure 11(a) shows the utility delivered to the client within the budget, using either of the page load strategies. For each (time budget, strategy) pair, we present a box and whiskers plot that shows the 10^{th}, 25^{th}, 50^{th}, 75^{th}, and 90^{th} percentiles across websites. We see that KLOTSKI consistently delivers a significantly better user experience. When user tolerance is 2 seconds, the fraction of high utility resources loaded within this limit on the median web page increases from 25% with the original website to roughly 60% with KLOTSKI. Similarly, we see KLOTSKI increasing the utility delivered on the median web page from 50% to almost 80% when the time budget is 3 seconds.

In addition, in Figure 11(b), we plot the distribution across websites of the difference between KLOTSKI and the original website in terms of the fraction of high utility resources loaded within the budget. KLOTSKI consistently fares better or no worse than the original website. For time budgets of 1–4 seconds, KLOTSKI manages to deliver an additional 20% of the high utility resources on roughly 20–40% of the websites.

Figure 12 illustrates these benefits offered by KLOTSKI by comparing the screenshots 3 seconds into the page load when loading an example website.



(a) (b) (c)

Figure 13: ***Comparison of utility delivered when (a) varying the function used to compute the utility values for high utility objects, (b) loading full versions of web pages (by faking a desktop's User-Agent), and (c) loading web pages on a desktop.***

We also compared the utility improvements offered by KLOTSKI to that obtained when loading web pages via a caching proxy. We consider the best case scenario where the proxy has cached all cacheable resources on every web page. However, we found that the user experience with a caching proxy is almost identical to that obtained with a proxy that simply relays communication between clients and webservers without any caching. Caching at the proxy does not offer any benefits because the 4G network between the client and the proxy is the bottleneck here. Hence, KLOTSKI's proactive delivery of static high utility content to the client is critical to enabling the improvements in user experience that it offers.

Impact of utility function: While we assigned a utility score of 1 to all high utility resources in the above experiment, one can also consider assigning different positive scores to different high utility resources. For example, among all above-the-fold resources, the user may derive larger utility from larger objects.

To evaluate the impact on KLOTSKI's benefits when varying the utility score across different high utility resources, we rerun the previous experiment with two utility functions. For any above-the-fold resource that is B bytes large and occupies an area A on the display, we assign a utility score of $\log_{10}(A)$ in one case and $\log_{10}(B)$ in the other case. For a time budget of 2 seconds, Figure 13(a) compares the improvement in user experience offered by KLOTSKI in these two cases as well as with the binary utility function that we used above. While the precise improvements vary across the utility functions, KLOTSKI improves the utility delivered on the median web page by over 60% in all three cases.

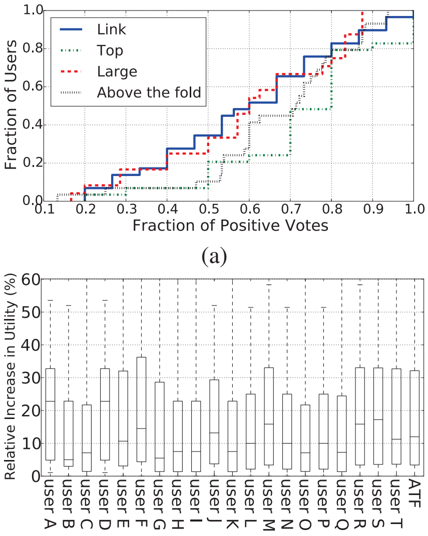
Utility for full versions of web pages and on desktops: Though our primary motivation in developing KLOTSKI is to improve user experience on the mobile web, its approach of reprioritizing important content can also be beneficial in other scenarios. For example, though many websites offer mobile-optimized versions, nearly a third of users prefer the full site experience [20] and 80% of mobile-generated revenue is generated when users view the full site [24]. However, page load times for these full versions are even worse than the poor performance on the average mobile-optimized web page. Similarly, though page load times are typically within 5 seconds on desktops (Figure 1(a)), recent surveys [11] show that 47% of users expect a page to load within 2 seconds and that 67% of users expect page loads on desktops to be faster than on mobile devices.

We evaluate KLOTSKI's ability to improve the web experience in these two scenarios by first loading full versions of web pages on a smartphone, and thereafter, by loading web pages on a desktop with a wired connection. We vary user tolerance from 2 to 5 seconds in the former case, and from 0.7 to 1.3 seconds in the latter. In both cases, we assign a utility score of 1 for all the above-the-fold resources and a score of 0 for other resources. Figure 13(b) and 13(c) show that KLOTSKI's reprioritization of important content helps significantly improve the user experience even in these cases.

Personalized preferences: So far, we considered all above-the-fold content important. We next evaluate KLOTSKI when accounting for user-specific preferences.

To capture user-specific utility preferences, we surveyed 120 users on Mechanical Turk. On our survey site (`http://object-study.appspot.com`), we show every visitor snapshots of 30 web pages—the landing pages of 30 websites chosen at random. For each page, we pick one resource on the page at random and ask the user to rate their perceived utility of each resource on a range varying from "Strong No" to "Strong Yes" (i.e., on a Likert scale from -2 to 2). We only consider data from respondents who 1) chose the correct rating for 4 objects known definitively to be very important or insignificant, and 2) gave consistent responses when they were asked again for their opinion on 5 (chosen at random) of the 30 objects that they had rated.

We observe significant variances in user preferences. For example, Figure 14 shows the distribution of utilities for four types of resources—has a link, in the top third of a page, larger than 100x100 pixels, or is above-the-fold. In each case, we see that the fraction of resources considered important ("Yes" or "Strong Yes") greatly varies across users. This validates the importance of KLOTSKI's approach of being able to account for arbitrary utility preferences, instead of existing approaches [14, 22] that can only optimize above-the-fold content.



(a)

(b)

Figure 14: ***(a) Variance across users of utility perceived in a few types of objects. (b) For 20 users, spread across websites of the difference between*** KLOTSKI ***and the original website in the fraction of high utility resources loaded within 2 seconds. ATF considers all above-the-fold content important.***

For a given user, we use her survey responses to estimate the utilities that she would perceive from the content on other web pages as follows. We partition all URLs into 9 categories, which are such that any given user's ratings are consistently positive or consistently negative across all the URLs that they rated in that category. For every (user, category) pair, we assign a utility score of 1/0 to all URLs in that category if a majority of the user's ratings for URLs in the category were positive/negative.

We consider the data gathered from 20 random users and evaluate KLOTSKI taking their preferences into account. For each of the 20 users, Figure 14 shows the distribution across websites of the difference between KLOTSKI and the native unmodified page load in terms of the fraction of high utility resources delivered within 2 seconds. For almost all users, we see that KLOTSKI increases the fraction of high utility resources loaded within 2 seconds by at least 20% on over 25% of websites. Moreover, most users see an increase as high as 50% on some websites. On the flip side, only few users see a worse experience on any website.

8.3 Evaluation of KLOTSKI's components

The improvement in user experience with KLOTSKI is made possible due to its combined use of several components. We evaluate each of these in isolation next.

8.3.1 Fingerprint generation

Matching replaced resources: First, we evaluate the accuracy with which the KLOTSKI back-end can map URL replacements across page loads. The primary challenge in doing so is that we do not have ground truth data (i.e., pairs of URLs which indeed replaced each other across page loads). It is very hard to identify matches manually, and no prior techniques exist for this task.

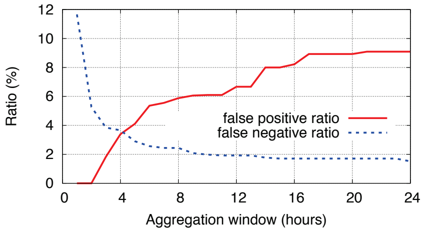


Figure 15: ***False positive/negative matches as a function of back-end's aggregation window for merging dependencies.***

Hence, we instead use the following evaluation strategy. We gathered a dataset wherein we fetched 500 web pages once every hour for a week. For every web page, we compare every pair of loads. As mentioned earlier, we find that our first technique for mapping replaced resources – *identical parent, lone replaced child* – is almost always accurate. Therefore, we consider all replacements identified using this technique as the ground truth, and use this data to evaluate our other two techniques for mapping replaced resources: *similar surrounding text* and *similar position on display*. When we apply these two techniques one followed by the other, we find that the matches obtained with a threshold of 100% for the local text similarity and 5 pixels for the display position similarity yield a 96% true positive rate and a 3% false positive rate. While local text similarity and display position similarity result in reasonably high false negative rates when applied in isolation, they enable accurate detection of URL replacements when used in combination.

Aggregation of dependency structures: Recall that KLOTSKI's back-end generates a fingerprint per web page by aggregating its measurements of that page over an aggregation window Δ, i.e., it aggregates measurements from Δ hours ago until now. Here, we ask: what should be the value of Δ? The smaller the value of Δ, the URL patterns stored in a page's dependency structure would not have converged sufficiently to capture the page's dynamics. The larger Δ, these URL patterns may become too generic, resulting in many false positive matches when the KLOTSKI front-end uses these patterns for dynamic prioritization of URLs.

We examine this trade-off with the same dataset as above where we loaded 500 pages once an hour for a week. After every hour, we applied the KLOTSKI back-end to generate a dependency structure for every page by aggregating measurements of that page over the past Δ hours. We then compute the number of false positive and false negative matches when using the patterns in the aggregated dependency structure to match URLs fetched in the next load of that page. Varying Δ from 1 to 24 hours,

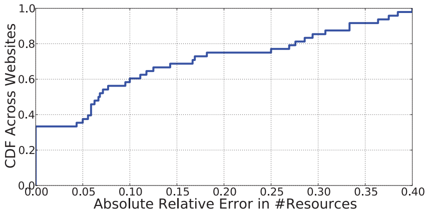


Figure 16: ***Comparison of no. of resources selected to load and no. of resources loaded in practice within the budget.***

Time budget (sec)	1	2	3	4	5
Median runtime (ms)	1	3	4	6	7

Table 1: ***Runtime of resource selection algorithm for the median web page as a function of time budget.***

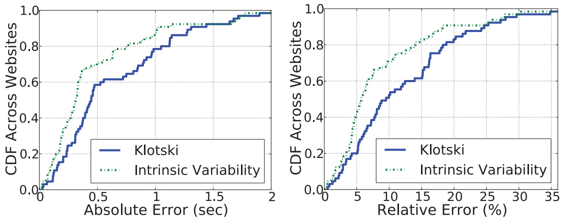
Figure 15 plots the false positive rate and false negative rate on the median web page. We see that an aggregation window of 4 hours presents the best trade-off.

8.3.2 Resource selection

Having already demonstrated the utility improvements offered by KLOTSKI, we now evaluate the correctness and efficiency of its selection of resources to prioritize.

First, we evaluate KLOTSKI's ability to accurately account for flux in page content when selecting resources. For every web page, we evaluate whether the number of resources selected by KLOTSKI's front-end for prioritization for a load time budget of 2 seconds match the number of high utility URLs received by the client within 2 seconds. These two values can differ due to errors in KLOTSKI's load time estimates and due to inaccuracies in KLOTSKI's dependency structure. Figure 16 plots the distribution across web pages of the absolute value of the relative difference between these two values. We see that our error is less than 20% on roughly 80% of websites (i.e., the number of resources delivered within the budget is within 20% of the number selected by the front-end), thus validating the correctness of KLOTSKI's fingerprints and the accuracy of its load time estimator.

Second, we examine the overhead of KLOTSKI's greedy resource selection algorithm. Recall that the execution of this algorithm is on the critical path of loading a web page, since the front-end begins executing the algorithm only when it receives the request for the page's main HTML. Table 1 shows that, across a range of budget values, the runtime of the front-end's resource selection is within 10 ms for the median web page. Given that the load time of the main HTML is itself greater than 500 ms on over 90% of web pages, combined with the fact that the average size of KLOTSKI's fingerprint for a web page is 1.6 KB (more than an order of magnitude lesser than the average size of the main HTML [12]), this shows that the front-end can fetch the fingerprint and finish executing the resource selection algorithm before it completes delivering the main HTML to the client.



(a) (b)

Figure 17: ***(a) Absolute and (b) relative errors in*** KLOTSKI***'s load time estimates. Comparison with intrinsic variability.***

8.3.3 Load time estimation

Finally, we evaluate the accuracy of KLOTSKI's load time estimator. We apply KLOTSKI to estimate the load time when web pages are loaded via the front-end, albeit without the front-end prioritizing any content. We compute the absolute and relative error in KLOTSKI's load time estimates compared to measured load times. Since a source of error here is the intrinsic variability in load times, we get a distribution of load time variability as follows. We load every web page 10 times and partition these loads into two sets of 5 loads each. For each page, we then compute the difference between the median measured load times in the two partitions. Figure 17 shows that the errors in KLOTSKI's load time estimates closely match the distribution of this intrinsic variability.

9 Discussion

Optimizing other metrics: Though KLOTSKI maximizes the utility delivered within a time budget, our design can easily accommodate other optimization criteria. For example, to help users cope with data caps, KLOTSKI's greedy algorithm can be modified to select a subset of resources that maximizes utility subject to a limit on the total number of bytes across the selected resources; all unselected resources can be blocked by the front-end. Similarly, given appropriate models of energy consumption, the front-end can deliver a subset of resources that keep energy consumed on the client within a limit.

Utility function: To obtain a user's preferences, we can have the user take a one-time survey (similar to our user study in Section 8.2) when she first begins using KLOTSKI. Alternatively, since going over several objects and rating them upfront can be onerous, KLOTSKI can initially start with a default utility function for every user, and on any web page that the user visits, we can provide the user the option of marking objects as low utility (say, whenever the page takes too long to load); e.g., the Adblock Plus browser extension similarly lets users mark ads that the user wants it to block in the future.

Measurement scheduling and personalized websites: Since websites differ in the rate of flux in their content, KLOTSKI's back-end can adaptively vary measurements of different web pages. For example, the back-end can more frequently load pages from news websites as compared to websites used for reference, since content in the former category changes more often than in the latter category. In addition, for any web page that personalizes content, the back-end can load the web page using different user accounts to capture which parts of the page stay the same across users and which parts change.

10 Related work

Our contribution here is in recognizing the need for, and presenting a practical solution for, improving mobile web experience by dynamically prioritizing content important to the user, rather than trying to reduce page load times. Here, we survey prior work related to KLOTSKI.

Measuring and modeling web performance: Prior efforts have analyzed the complexity of web pages [12, 17, 33] and how it impacts page load times [33, 60, 34] and energy consumption [58, 31, 55]. Our work is motivated by such measurements.

Characterizing webpage content: Song et al. [57] develop techniques to partition webpages into logical blocks to identify blocks that users consider important. Other related efforts compare different pages in the same website [62, 43] or use the DOM tree [40]. Unlike these previous efforts, KLOTSKI associates utilities with individual resources rather than blocks. The closest works on dependency inference are WebProphet [42] and WProf [58]. KLOTSKI infers a more high-level structure robust to the flux in content across loads.

Better browsers and protocols: There are several proposals for new web protocols (e.g., SPDY [25]), guidelines for optimizing pages (e.g., [9]), and optimized browsers (e.g., [45, 44, 21]) and hardware [47, 63]. Many studies have shown that these do not suffice, e.g., two recent studies [59, 37] show that SPDY-like optimizations do not improve performance significantly and interact poorly with cellular networks. KLOTSKI pursues a complementary approach to prioritize important resources.

Cloud-based mobile acceleration: KLOTSKI's architecture is conceptually similar to recent cloud-based mobile web acceleration services (e.g., [2, 21]). A recent study suggests that these can hurt performance [56]. The key difference is that our objective is to maximize user-perceived utility rather than optimize page load times.

Web prefetching: A widely studied approach for improving web performance on mobile devices has been to prefetch content [50, 41, 38]. However, despite the large body of work on accurately predicting what content should be prefetched [51, 49, 36], prefetching is rarely used in practice on mobile devices due to the overheads on energy and data usage imposed by prefetching content that is never used [53]. KLOTSKI's approach of pushing high utility resources on a web page to a client only once the client initiates the load of that page improves user experience without delivering unnecessary content.

Prioritizing important content: Concurrent to our work, some startups (e.g., InstartLogic and StrangeLoop Networks) try to deliver higher priority resources earlier. Based on public information [29, 20], these appear to optimize certain types of content such as images and Flash, and do not incorporate user preferences like KLOTSKI. We are not aware of published work that highlights how they address the challenges w.r.t. dependencies, optimization, and load time estimation that we tackle. Moreover, their approach requires website providers to use their CDN services, whereas KLOTSKI does not explicitly require any changes to web providers.

Older efforts that dynamically re-order the delivery of web content are limited to prioritizing above-the-fold resources [14, 35]. Based on the observation from our study that users significantly differ in the content that they consider important on the same page, we instead design and implement KLOTSKI to account for arbitrary utility functions.

11 Conclusions

Our work tackles a set of contradictory trends in the mobile web ecosystem today – users desire rich content but have decreasing tolerance, even as current performance optimizations yield low returns due to increasing website complexity. In light of these trends, KLOTSKI takes the stance that rather than blindly try to improve performance, we should try to dynamically reprioritize the delivery of a web page to deliver higher utility content within user tolerance limits.

We addressed several challenges to realize this approach in practice: dependencies across content on a page, complexity of the optimization, difficulty in estimating load times, and delivering benefits with minimal changes to clients and webservers. Our evaluation shows that KLOTSKI's algorithms tackle these challenges effectively and that it yields up to a 60% increase in user-perceived utility. While our focus was on the imminent challenge of improving mobile web user experiences, the ideas in KLOTSKI are more broadly applicable to other scenarios (e.g., desktop) and requirements (e.g., energy).

Acknowledgment

We thank Google's support of this work in the form of a Faculty Research Award.

References

[1] 2012 state of mobile ecommerce performance. http://www.strangeloopnetworks.com/resources/research/state-of-mobile-ecommerce-performance.

[2] Amazon Silk. http://amazonsilk.wordpress.com/.

[3] Amazon's Silk browser acceleration tested: Less bandwidth consumed, but slower performance. http://www.anandtech.com/show/5139/amazons-silk-browser-tested-less-bandwidth-consumed-but-slower-performance.

[4] Average number of web page objects breaks 100. http://www.websiteoptimization.com/speed/tweak/average-number-web-objects/.

[5] Average web page size triples since 2008. http://www.websiteoptimization.com/speed/tweak/average-web-page/.

[6] The Chromium blog: Experimenting with QUIC. http://blog.chromium.org/2013/06/experimenting-with-quic.html.

[7] Convert any site into mobile format with Google Mobilizer. http://www.geek.com/articles/mobile/convert-any-site-into-mobile-format-with-google-mobilizer-20060117/.

[8] Data compression proxy. https://developer.chrome.com/multidevice/data-compression.

[9] Google page speed. http://code.google.com/speed/page-speed/.

[10] Google research: No mobile site = lost customers. http://www.forbes.com/sites/roberthof/2012/09/25/google-research-no-mobile-site-lost-customers/.

[11] How loading time affects your bottom line. https://blog.kissmetrics.com/loading-time/.

[12] HTTP archive. http://httparchive.org/.

[13] Hypertext transfer protocol version 2. https://http2.github.io/http2-spec/.

[14] Imageloader utility: Loading images below the fold. http://www.yuiblog.com/yui/md5/yui/2.8.0r4/examples/imageloader/imgloadfold.html.

[15] Keynote: Mobile retail performance index. http://www.keynote.com/performance-indexes/mobile-retail-us.

[16] Keynote systems. http://www.keynote.com.

[17] Let's make the web faster. http://code.google.com/speed/articles/web-metrics.html.

[18] Mobify. http://mobify.me.

[19] Mobile Internet will soon overtake fixed Internet. http://gigaom.com/2010/04/12/mary-meeker-mobile-internet-will-soon-overtake-fixed-internet/.

[20] Mobile site optimization. http://www.strangeloopnetworks.com/assets/PDF/downloads/Strangeloop-Site-Optimizer-Whitepaper.pdf.

[21] Opera Mini & Opera Mobile browsers. http://www.opera.com/mobile/.

[22] PageSpeed service: Cache and prioritize visible content. https://developers.google.com/speed/pagespeed/service/PrioritizeAboveTheFold.

[23] Pew research Internet project: Cell Internet use 2013. http://www.pewinternet.org/2013/09/16/cell-internet-use-2013/.

[24] Rules for mobile performance optimization. http://queue.acm.org/detail.cfm?id=2510122.

[25] SPDY: An experimental protocol for a faster web. http://www.chromium.org/spdy/spdy-whitepaper.

[26] SPDY coming to Safari, future versions of IE. http://zoompf.com/blog/2014/06/spdy-coming-to-safari-future-versions-of-ie.

[27] SPDY server for node.js. https://github.com/indutny/node-spdy.

[28] StatCounter global stats. http://gs.statcounter.com/#desktop+mobile+tablet-comparison-ww-monthly-201309-201408.

[29] Web application streaming: A radical new approach. http://go.instartlogic.com/rs/growthfusion/images/Updated_White_Paper_Instart_Logic.pdf.

[30] What users want from mobile. http://www.slideshare.net/Gomez_Inc/2011-mobile-survey-what-users-want-from-mobile.

[31] L. Bent and G. M. Voelker. Whole page performance. In *WCW*, 2002.

[32] A. Bouch, A. Kuchinsky, and N. Bhatti. Quality is in the eye of the beholder: Meeting users' requirements for internet quality of service. In *CHI*, 2000.

[33] M. Butkiewicz, H. V. Madhyastha, and V. Sekar. Understanding website complexity: Measurements, metrics, and implications. In *IMC*, 2011.

[34] M. Butkiewicz, Z. Wu, S. Li, P. Murali, V. Hris-

tidis, H. V. Madhyastha, and V. Sekar. Enabling the transition to the mobile web with WebSieve. In *HotMobile*, 2013.

[35] T.-Y. Chang, Z. Zhuang, A. Velayutham, and R. Sivakumar. Client-side web acceleration for low-bandwidth hosts. In *BROADNETS*, 2007.

[36] X. Chen and X. Zhang. A popularity-based prediction model for web prefetching. *IEEE Computer*, 36(3):63–70, 2003.

[37] J. Erman, V. Gopalakrishnan, R. Jana, and K. Ramakrishnan. Towards a SPDY'ier mobile web. In *CoNEXT*, 2013.

[38] L. Fan, P. Cao, W. Lin, and Q. Jacobson. Web prefetching between low-bandwidth clients and proxies: Potential and performance. In *SIGMETRICS*, 1999.

[39] D. Galletta, R. Henry, S. McCoy, and P. Polak. Web site delays: How tolerant are users? *Journal of the Association for Information Systems*, 2004.

[40] S. Gupta, G. Kaiser, D. Neistadt, and P. Grimm. DOM-based content extraction of HTML documents. In *WWW*, 2003.

[41] Z. Jiang and L. Kleinrock. Web prefetching in a mobile environment. *IEEE Personal Communications*, 5(5):25–34, 1998.

[42] Z. Li, M. Zhang, Z. Zhu, Y. Chen, A. Greenberg, and Y.-M. Wang. WebProphet: Automating performance prediction for web services. In *NSDI*, 2010.

[43] S. H. Lin and J. M. Ho. Discovering informative content blocks from web documents. In *KDD*, 2002.

[44] D. Lymberopoulos, O. Riva, K. Strauss, A. Mittal, and A. Ntoulas. PocketWeb: Instant web browsing for mobile devices. In *ASPLOS*, 2012.

[45] H. Mai, S. Tang, S. T. King, C. Cascaval, and P. Montesinos. A case for parallelizing web pages. In *HotPar*, 2012.

[46] R. Margolies, A. Sridharan, V. Aggarwal, R. Jana, N. K. Shankaranarayanan, V. A. Vaishampayan, and G. Zussman. Exploiting mobility in proportional fair cellular scheduling: Measurements and algorithms. In *INFOCOM*, 2014.

[47] L. Meyerovich and R. Bodik. Fast and parallel web page layout. In *WWW*, 2010.

[48] F. Nah. A study on tolerable waiting time: How long are Web users willing to wait? *Behaviour & Information Technology*, 23(3), May 2004.

[49] A. Nanopoulos, D. Katsaros, and Y. Manolopoulos. A data mining algorithm for generalized web prefetching. *IEEE Transactions on Knowledge and Data Engineering*, 15(5):1155–1169, 2003.

[50] V. N. Padmanabhan and J. C. Mogul. Using predictive prefetching to improve world wide web latency. *ACM SIGCOMM Computer Communication Review*, 26(3):22–36, 1996.

[51] T. Palpanas. Web prefetching using partial match prediction. Technical Report CSRG 376, University of Toronto, 1998.

[52] J. W. Ratcliff and D. E. Metzener. Pattern matching: The gestalt approach. *Dr Dobbs Journal*, 13(7):46, 1988.

[53] L. Ravindranath, S. Agarwal, J. Padhye, and C. Riederer. Give in to procrastination and stop prefetching. In *HotNets*, 2013.

[54] P. Romirer-Maierhofer, F. Ricciato, A. D'Alconzo, R. Franzan, and W. Karner. Network-wide measurements of TCP RTT in 3G. In *Traffic Monitoring and Analysis*, 2009.

[55] A. Sampson, C. Cascaval, L. Ceze, P. Montesinos, and D. S. Gracia. Automatic discovery of performance and energy pitfalls in HTML and CSS. In *IISWC*, 2012.

[56] A. Sivakumar, V. Gopalakrishnan, S. Lee, S. Rao, S. Sen, and O. Spatscheck. Cloud is not a silver bullet: A case study of cloud-based mobile browsing. In *HotMobile*, 2014.

[57] R. Song, H. Liu, J.-R. Wen, and W.-Y. Ma. Learning block importance models for web pages. In *WWW*, 2004.

[58] X. S. Wang, A. Balasubramanian, A. Krishnamurthy, and D. Wetherall. Demystifying page load performance with WProf. In *NSDI*, 2013.

[59] X. S. Wang, A. Balasubramanian, A. Krishnamurthy, and D. Wetherall. How speedy is SPDY? In *NSDI*, 2014.

[60] Z. Wang, F. X. Lin, L. Zhong, and M. Chishtie. Why are web browsers slow on smartphones? In *HotMobile*, 2011.

[61] Y. Xie, F. Yu, K. Achan, R. Panigrahy, G. Hulten, and I. Osipkov. Spamming botnets: Signatures and characteristics. In *SIGCOMM*, 2008.

[62] L. Yi and B. Liu. Eliminating noisy information in web pages for data mining. In *KDD*, 2003.

[63] Y. Zhu and V. J. Reddi. High-performance and energy-efficient mobile web browsing on big/little systems. In *HPCA*, 2013.

Information-Agnostic Flow Scheduling for Commodity Data Centers

Wei Bai[1], Li Chen[1], Kai Chen[1], Dongsu Han[2], Chen Tian[3], Hao Wang[1]
[1]*SING Group @ HKUST* [2]*KAIST* [3]*Nanjing Univ.*

Abstract

Many existing data center network (DCN) flow scheduling schemes minimize flow completion times (FCT) based on prior knowledge of flows and custom switch functions, making them superior in performance but hard to use in practice. By contrast, we seek to minimize FCT with no prior knowledge and existing commodity switch hardware.

To this end, we present PIAS[1], a DCN flow scheduling mechanism that aims to minimize FCT by mimicking Shortest Job First (SJF) on the premise that flow size is not known *a priori*. At its heart, PIAS leverages multiple priority queues available in existing commodity switches to implement a Multiple Level Feedback Queue (MLFQ), in which a PIAS flow is gradually demoted from higher-priority queues to lower-priority queues based on the number of bytes it has sent. As a result, short flows are likely to be finished in the first few high-priority queues and thus be prioritized over long flows in general, which enables PIAS to emulate SJF without knowing flow sizes beforehand.

We have implemented a PIAS prototype and evaluated PIAS through both testbed experiments and ns-2 simulations. We show that PIAS is readily deployable with commodity switches and backward compatible with legacy TCP/IP stacks. Our evaluation results show that PIAS significantly outperforms existing information-agnostic schemes. For example, it reduces FCT by up to 50% and 40% over DCTCP [11] and L2DCT [27] respectively; and it only has a 4.9% performance gap to an ideal information-aware scheme, pFabric [13], for short flows under a production DCN workload.

1 Introduction

There has been a virtually unanimous consensus in the community that one of the most important goals for data center network (DCN) transport designs is to minimize the flow completion times (FCT) [11–13, 22, 26, 27]. This is because many of today's cloud applications, such as web search, social networking, and retail recommendation, have very demanding latency requirements, and even a small delay can directly affect application performance and degrade user experience [11, 27].

To minimize FCT, most recent proposals [13, 22, 26, 34] assume prior knowledge of accurate per-flow information, e.g., flow sizes or deadlines, to achieve superior performance. For example, PDQ, pFabric and PASE [13, 22, 26] all assume flow size is known *a priori*, and attempt to approximate Shortest Job First (SJF, preemptive), which is the optimal scheduling discipline for minimizing the average FCT over a single link. In this paper, we question the validity of this assumption, and point out that, for many applications, such information is difficult to obtain, and may even be unavailable (§2). Existing transport layer solutions with this assumption is therefore very hard to implement in practice.

We take one step back and ask: without prior knowledge of flow size information, what is the best scheme that minimizes FCT with existing commodity switches?

Motivated by the above question, we list our key design goals as follows:

- **Information-agnostic:** Our design must not assume *a priori* knowledge of flow size information being available from the applications.
- **FCT minimization:** The solution must be able to enforce an optimal information-agnostic flow scheduling. It should minimize average and tail FCTs for latency-sensitive short flows, while not adversely affecting the FCTs of long flows.
- **Readily-deployable:** The solution must work with existing commodity switches in DCNs and be backward compatible with legacy TCP/IP stacks.

When exploring possible solution spaces, we note that some existing approaches such as DCTCP, HULL, L2DCT, etc [11, 12, 27] reduce FCT without relying on flow size information. They generally improve FCT by maintaining low queue occupation through mechanisms like adaptive congestion control, ECN, pacing, etc. However, they do not provide a full answer to our question, because they mainly perform end-host based rate control which is ineffective for flow scheduling [13, 26].

[1] PIAS, Practical Information-Agnostic flow Scheduling, was first introduced in an earlier workshop paper [14] which sketched a preliminary design and the initial results.

In this paper, we answer the questions with PIAS, a practical information-agnostic flow scheduling that minimizes the FCT in DCNs. In contrast to previous FCT minimization schemes [13, 22, 26] that emulate SJF by using prior knowledge of flow sizes, PIAS manages to mimic SJF with no prior information. At its heart, PIAS leverages multiple priority queues available in existing commodity switches to implement a Multiple Level Feedback Queue (MLFQ), in which a PIAS flow is gradually demoted from higher-priority queues to lower-priority queues based on the bytes it has sent during its lifetime. In this way, PIAS ensures in general that short flows are prioritized over long flows, effectively emulating SJF without knowing flow sizes beforehand.

However, we face several concrete challenges to make PIAS truly effective. First, how to determine the demoted threshold for each queue of MLFQ? Second, as flow size distribution varies across time and space, how to keep PIAS's performance in such a dynamic environment? Third, how to ensure PIAS's compatibility with legacy TCP/IP stacks in production DCNs?

For the first challenge, we address it by deriving a set of optimal demotion thresholds for MLFQ through solving a FCT minimization problem. We further show that the derived threshold setting is robust to a reasonable range of traffic distributions. This encourages us to distribute the thresholds to the end hosts for packet tagging while only performing strict priority queueing, a built-in function, in the PIAS switches.

For the second challenge, as one set of demotion thresholds works the best for a certain range of traffic distributions, PIAS adjusts thresholds to keep up with traffic dynamics. However, the key problem is that a mismatch between thresholds and underlying traffic is inevitable. Once that happens, short flows may be adversely affected by large ones in a queue, impacting on their latency. Inspired by ideas from ECN-based rate control [11], PIAS employs ECN to mitigate our mismatch problem. Our reasoning is that, by maintaining low queue occupation, short flows always see small queues and thus will not be seriously delayed even if they are mistakenly placed in a queue with a long flow due to mismatched thresholds.

For the third challenge, we employ DCTCP-like transport at the PIAS end hosts and find that PIAS interacts favorably with DCTCP or other legacy TCP protocols with ECN enabled. A potential problem is that many concurrent short flows may starve a coexisting long flow, triggering TCP timeouts and degrading application performance. We measure the extent of starvation on our testbed with a realistic workload and analyze possible solutions. We ensure that all mechanisms in PIAS can be implemented by a shim layer over NIC without touching the TCP stack.

We have implemented a PIAS prototype (§4). On the end host, we implement PIAS as a kernel module in Linux, which resides between the Network Interface Card (NIC) driver and the TCP/IP stack as a shim layer. It does not touch any TCP/IP implementation that natively supports various OS versions. In virtualized environments, PIAS can also support virtual machines well by residing in hypervisor (or Dom 0). On the switch, PIAS only needs to enable priority queues and ECN which are both built-in functions readily supported by existing commodity switch hardware.

We evaluate PIAS on a small-scale testbed with 16 Dell servers and a commodity Pronto-3295 Gigabit Ethernet switch (Broadcom BCM#56538). In our experiments, we find that PIAS reduces the average FCT for short flows by ∼37-47% and ∼30-45% compared to DCTCP under two realistic DCN traffic patterns. It also improves the query performance by ∼28-30% in a Memcached [7] application (§5.1). We further dig into different design components of PIAS such as queues, optimal demotion threshold setting, ECN, and demonstrate the effectiveness of each of their contributions to PIAS's performance (§5.2).

To complement our small-scale testbed experiments, we further conduct large-scale simulations in a simulated 10/40G network using ns-2 [10]. In our simulations, we show that PIAS outperforms all existing information-agnostic solutions under realistic DCN workloads, reducing the average FCT for short flows by up to 50% and 40% compared to DCTCP and L2DCT respectively. In addition, our results show that PIAS, as a readily-deployable information-agonistic scheme, also delivers a comparable performance to a clean-slate information-aware design, pFabric [13], in certain scenarios. For example, there is only a 4.9% gap to pFabric for short flows in a data mining workload [21] (§5.3).

To make our work easy to reproduce, we made our implementation and evaluation scripts available online at: `http://sing.cse.ust.hk/projects/PIAS`.

2 Motivation

To motivate our design, we introduce a few cases in which the accurate flow size information is hard to obtain, or simply not available.

HTTP chunked transfer: Chunked transfer has been supported since HTTP 1.1 [20], where dynamically generated content is transferred during the generation process. This mode is widely used by datacenter applications. For example, applications can use chunked transfer to dump database content into OpenStack Object Storage [9]. In chunked transfer, a flow generally consists of multiple chunks, and the total flow size is not available at the start of the transmission.

Database query response: Query response in database systems, such as Microsoft SQL Server [8], is another example. Typically, SQL servers send partial query results as they are created, instead of buffering the result until the end of the query execution process [8]. The flow size again is not available at the start of a flow.

Stream processing: Stream processing systems are currently gaining popularity. In Apache Storm [2], after the master node distributes tasks to worker nodes, workers will analyze the tasks and pre-establish persistent connections with related worker nodes. During the data processing, data tuples completed in one node are continuously delivered to the next node in the stream processing chain. The amount of data to be processed is unknown until the stream finishes.

Practical limitations: We note that there are certain cases where the flow size information can be obtained or inferred. For example, in Hadoop [5] the mapper will first write the intermediate data to disk before the corresponding reducer starts to fetch the data, thus the flow size can be obtained in advance [28]. Even so, practical implementation issues are still prohibitive. First, we need to patch all modules in every application that generate network traffic, which is a burden for applications programmers and/or network operators. Second, current operating systems lack appropriate interface for delivering the flow size information to the transport layer. Thus, kernel modifications are also required.

3 The PIAS Design

3.1 Design Rationale

Compared to previous solutions [13, 22, 26] that emulate SJF based on prior knowledge of flow sizes, PIAS distinguishes itself by emulating SJF with no prior information. At its core, PIAS exploits multiple priority queues available in commodity switches and implements a MLFQ, in which a PIAS flow is demoted from higher-priority queues to lower-priority queues dynamically according to its bytes sent. Through this way, PIAS enables short flows to finish in the first few priority queues, and thus in general prioritizes them over long flows, effectively mimicking SJF without knowing the flow sizes.

We note that scheduling with no prior knowledge is known as non-clairvoyant scheduling [25]. Least Attained Service (LAS) is one of the best known algorithms that minimize the average FCT in this case [30]. LAS tries to approximate SJF by guessing the remaining service time of a job based on the service it has attained so far. LAS is especially effective in DCN environments where traffic usually exhibits long-tail distribution—most flows are short and a small percent are very large [11, 21].

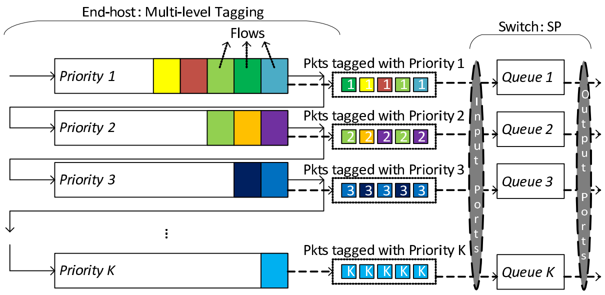


Figure 1: PIAS overview

PIAS is partially inspired by LAS. However, we find that directly enabling LAS on switches requires us to compare the amount of bytes transferred for each flow, which is not supported in existing commodity switches. Furthermore, although DCN traffic distribution is generally long-tailed, it varies across both time and space, and on some switch ports the distribution may temporarily not be so. Blindly using LAS will exacerbate the problem when multiple long flows coexist on a port, as pure LAS favors short flows but performs badly when a long flow meets a longer flow, causing the longer one to starve.

To this end, PIAS leverages multiple priority queues available in existing commodity switches (typically 4–8 queues per port [13]) to implement a MLFQ (see Figure 1). Packets in different queues of MLFQ are scheduled with strict priority, while packets in the same queue are scheduled based on FIFO. In a flow's lifetime, it is demoted dynamically from ith queue down to the $(i+1)$th queue after transmitting more bytes than queue i's demotion threshold, until it enters the last queue. To further prevent switches from maintaining the per-flow state, PIAS distributes packet priority tagging (indicating a flow's sent size) to end hosts, allowing the PIAS switches to perform strict priority queueing only, which is already a built-in function in today's commodity switches.

By implementing MLFQ, PIAS gains two benefits. First, it prioritizes short flows over large ones because short flows are more likely to finish in the first few higher priority queues while large flows are eventually demoted to lower priority queues. This effectively enables PIAS to approximate SJF scheduling that optimizes average FCT while being readily implementable with existing switch hardware. Second, it allows large flows that are demoted to the same low priority queues to share the link fairly. This helps to minimize the response time of long flows, mitigating the starvation problem.

However, there are several concrete challenges to consider in order to make PIAS truly effective. First, how to determine the demotion threshold for each queue of MLFQ to minimize the FCT? Second, as DCN traffic varies across both time and space, how to make PIAS perform efficiently and stably in such a dynamic environment? Third, how to ensure PIAS's compatibility with legacy TCP/IP stacks in production DCNs? Next, we explain the details of the mechanism we design to address all these challenges.

3.2 Detailed Mechanisms

At a high level, PIAS's main mechanisms include distributed packet tagging, switch design, and rate control.

3.2.1 Packet Tagging at End-hosts

PIAS performs distributed packet tagging at end hosts as shown in Figure 1. There are K priorities $P_i, 1 \leq i \leq K$ and $(K-1)$ demotion thresholds $\alpha_j, 1 \leq j \leq K-1$. We assume $P_1 > P_2 ... > P_K$ and $\alpha_1 \leq \alpha_2 ... \leq \alpha_{K-1}$.

At the end host, when a new flow is initialized, its packets will be tagged with the highest priority P_1, giving it the highest priority in the network. As more bytes are sent, the packets of this flow will be tagged with decreasing priorities P_j $(2 \leq j \leq K)$ and enjoy decreasing priorities in the network[2]. The threshold to demote priority from P_{j-1} to P_j is α_{j-1}.

One challenge is to determine the demotion threshold for each priority to minimize the average FCT. By solving a FCT minimization problem, we derive a set of analytical solutions for optimal demotion thresholds (details in §3.3). Note that in PIAS we calculate the thresholds based on traffic information from the entire DCN and distribute the same threshold setting to all the end hosts. Our experiments and analysis show that such threshold setting is effective and also robust to a certain range of traffic variations (§5). This is a key reason we can decouple packet tagging from switches to end hosts while still maintaining good performance, which relieves the PIAS switches of having to keep the per-flow state.

As traffic changes over time, PIAS need to adjust the demotion thresholds accordingly. To keep track of traffic variations, each end host can periodically report its local traffic information to a central entity for statistics and many existing techniques exist for this purpose [37]. However, historical traffic may not reflect the future perfectly. Mismatches between threshold setting and underlying traffic are inevitable, which can hurt latency sensitive short flows. Therefore, mitigating the impact of the mismatch is a must for PIAS to operate in the highly dynamic DCNs. Our solution, as shown subsequently, is to employ ECN.

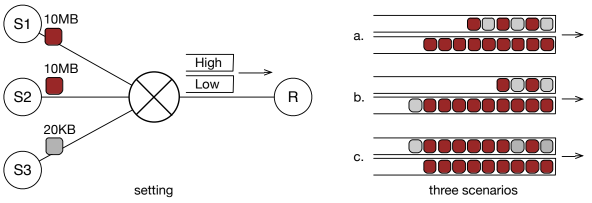
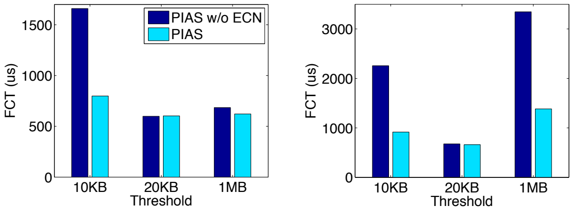


Figure 2: Illustration example: (a) threshold right; (b) threshold too small, packets of short flow get delayed by long flow after prematurely demoted to the low priority queue; (c) threshold too large, packets of large flow stay too long in the high priority queue, affecting short flow.



(a) Mean (b) 99th Percentile

Figure 3: Completion time of 20KB short flows

3.2.2 Switch Design

The PIAS switches enable the following two basic mechanisms, which are built-in functions for existing commodity switches [26].

- Priority scheduling: Packets are dequeued based on their priorities strictly when a fabric port is idle.
- ECN marking: The arriving packet is marked with Congestion Experienced (CE) if the instant buffer occupation is larger than the marking threshold.

With priority scheduling at the switches and packet tagging at the end hosts, PIAS performs MLFQ-based flow scheduling on the network fabric with stateless switches. Packets with different priority tags are classified into different priority queues. When the link is idle, the head packet from the highest non-empty priority queue is transmitted.

One may wonder why weighted fair queueing (WFQ) is not used to avoid starvation for long-lived flows. However, we choose priority scheduling for two reasons. First, priority queueing can provide better in-network prioritization and potentially achieve lower FCT than WFQ. Second, WFQ may cause packet out-of-order problem, thus degrading TCP performance.

Our intention to employ ECN is to mitigate the effect of the mismatch between the demotion thresholds and the traffic distribution. We use a simple example to illustrate the problem and the effectiveness of our solution. We connect 4 servers to a Gigabit switch as in Figure 2. One server is receiver (R) and the other three are senders (S1, S2 and S3). In our experiment, the receiver R continuously fetches 10MB data from S1 and S2, and 20KB data from S3. We configure the strict priority queueing

[2] In certain cases, applications may build persistent TCP connections to keep delivering request-response short messages for a long time. These persistent connections will eventually be assigned to the lowest priority due to the large cumulative size of bytes sent. To address this, we can periodically reset flow states based on more behaviors of traffic. For example, when a flow idles or keeps a very low average throughput for some time, we may reset the bytes sent from this flow to 0.

with 2 queues on the switch egress port to R. Since there are two priorities, we only have one demotion threshold from the high priority queue to the low priority queue. For this case, the optimal demotion threshold should be 20KB (achieving SJF in effect).

We intentionally apply three different thresholds 10KB, 20KB and 1MB, and measure the FCT of the 20KB short flows. Figure 3 shows the results of PIAS and PIAS without ECN. When the threshold is 20KB, both PIAS and PIAS without ECN achieve an ideal FCT. However, with a larger threshold (1MB) or a smaller threshold (10KB), PIAS shows obvious advantages over PIAS without ECN at both the average and 99th percentile. This is because, if the threshold is too small, packets of short flows prematurely enter the low priority queue and experience queueing delay behind long flows (see scenario (b)); if the threshold is too large, packets of long flows over-stay in the high priority queue, also affecting the latency of short flows (see scenario (c)).

By employing ECN, we can keep low buffer occupation and minimize the impact of long flows on short flows, which makes PIAS more robust to the mismatch between the demotion thresholds and traffic distribution.

3.2.3 Rate Control

PIAS employs DCTCP [11] as end host transport, and other legacy TCP protocols with ECN enabled can also be integrated into PIAS. We require PIAS to interact smoothly with the legacy TCP stack. One key issue is to handle flow starvation: when packets of a large flow get starved in a low priority queue for long time, this may trigger TCP timeouts and retransmissions. The frequent flow starvation may disturb the transport layer and degrade application performance. For example, a TCP connection which is starved for long time may be terminated unexpectedly.

To address the problem, we first note that PIAS can well mitigate the starvation between long flows, because two long flows in the same low priority queue will fairly share the link in a FIFO manner. In this way, PIAS minimizes the response time of each long flow, effectively eliminating TCP timeouts.

However, it is still possible that many concurrent short flows will starve a long flow, triggering its TCP timeouts. To quantify the effect, we run the web search benchmark traffic [11] (Figure 5) at 0.8 load in our 1G testbed, which has 16 servers connected to a Gigabit switch. We set RTOmin to 10ms and allocate 8 priority queues for PIAS. This experiment consists of 5,000 flows (around 5.7 million MTU-sized packets). We enable both ECN and dynamic buffer management in our switch. Hence, TCP timeouts are mainly caused by starvation rather than packet drops. We measure the number of TCP timeouts to quantify the extent of the starvation. We find that there are only 200 timeout events and 31 two consecutive timeout events in total. No TCP connection is terminated unexpectedly. The results indicate that, even at a high load, flow starvation is not common and will not degrade application performance adversely. We believe one possible reason is that the per-port ECN we used (see §4.1.2) may mitigate starvation by pushing back high priority flows when many packets from low priority long flows get starved. Another possible solution for handling flow starvation is treating a long-term starved flow as a new flow. For example, if a flow experiences two consecutive timeouts, we set its bytes sent back to zero. This ensures that a long flow can always make progress after timeouts. Note that the implementation of the above mechanism can be integrated to our packet tagging module without any changes to the networking stack.

Note that PIAS is free of packet reordering. This is because, during its lifetime, a PIAS flow is always demoted from a higher priority queue to a lower priority queue. In this way, an earlier packet is guaranteed to dequeue before a latter packet at each hop.

3.2.4 Discussion

Local decision: The key idea of PIAS is to emulate SJF which is optimal to minimize average FCT over a single link. However, there does not exist an optimal scheduling policy to schedule flows over an entire DCN with multiple links [13]. In this sense, similar to pFabric [13], PIAS also makes switch local decisions. This approach in theory may lead to some performance loss over the fabric [26]. For example, when a flow traverses multiple hops and gets dropped at the last hop, it causes bandwidth to be wasted on the upstream links that could otherwise have been used to schedule other flows. We note that some existing solutions [22, 26] leverage arbitration, where a common network entity allocates rates to each flow based on global network visibility, to address this problem. However, it is hard to implement because it requires non-trivial switch changes [22] or a complex control plane [26], which is against our design goal. Fortunately, local-decision based solutions maintain very good performance for most scenarios [13] and only experience performance loss at extremely high loads, e.g., over 90% [26]. However, most DCNs operate at moderate loads, e.g., 30% [16]. Our ns-2 simulation (§5.3) with production DCN traffic further confirms that PIAS works well in practice.

Demotion threshold updating: By employing ECN, PIAS can effectively handle the mismatch between the demotion thresholds and traffic distribution (see §5.2). This suggests that we do not need to frequently change our demotion thresholds which may be an overhead. In this paper, we simply assume the demotion thresholds are updated periodically according to the network scale (which decides the time for information collection and distribution) and leave dynamic threshold updates as future work.

3.3 Optimizing Demotion Thresholds

In this section, we describe our formulation to derive the optimal demotion thresholds for minimizing the average FCT. We identify this problem as a Sum-of-Linear-Ratios problem and provide a method to derive the optimal thresholds analytically for *any* given load and flow size distribution. We find that the demotion thresholds depend on both load and flow size distribution. As flow size distribution and load change across both time and space, ideally, one should use different thresholds for different links at different times. However, in practice, it is quite challenging to obtain such fine-grained link level traffic information across the entire DCN. Hence, we use the overall flow size distribution and load measured in the entire DCN as an estimate to derive a common set of demotion thresholds for all end hosts. We note that this approach may not be theoretically optimal and there is room for improvement. However, it is more practical and provides considerable gains, as shown in the evaluation (§5).

Problem formulation: We assume there are K priority queues $P_i(1{\le}i{\le}K)$ where P_1 has the highest priority. We denote the threshold for demoting the priority from $j{-}1$ to j as $\alpha_{j-1}(2{\le}j{\le}K)$. We define $\alpha_K{=}\infty$, so that the largest flows are all in this queue, and $\alpha_0{=}0$. The flows, indexed from $i{=}1$ to N, have sizes x_i. Denote the cumulative density function of flow size distribution as $F(x)$, thus $F(x)$ is the probability that a flow size is no larger than x. Note that we do not assume the any specific properties of $F(x)$, and it is used for convenience for the following derivation.

Let $\theta_j{=}F(\alpha_j){-}F(\alpha_{j-1})$, the percentage of flows with sizes in $[\alpha_{j-1},\alpha_j)$. For a flow with size in $[\alpha_{j-1},\alpha_j)$, it experiences the delays in different priorities up to the j-th priority. Denote T_j as the average time spent in the j-th queue. For a flow with size x, let x^+ be the residual size of this flow tagged with its lowest priority. Thus, $x^+{=}x{-}\alpha_{max}(x)$, where $\alpha_{max}(x)$ is the largest demotion threshold less than x, and let $j_{max}(x)$ be the index of this threshold.

So the average FCT for this flow is: $T(x){=}\sum_{l=1}^{j_{max}(x)}T_l{+}\frac{x^+}{1-\rho_{j_{max}(x)}}$.

The second term is bounded by $T_{j_{max}(x)}$, thus an upper bound is therefore: $T(x){\le}\sum_{l=1}^{min(j_{max}(x)+1,K)}T_l$.

We have the following optimization problem, where we choose an optimal set of thresholds $\{\alpha_j\}$ to minimize the objective: the average FCT of flows on this bottleneck link:

$$\begin{aligned}\min_{\{\alpha_j\}} \quad & \mathcal{T}{=}\sum_{l=1}^{K}(\theta_l\sum_{m=1}^{l}T_m){=}\sum_{l=1}^{K}(T_l\sum_{m=l}^{K}\theta_m) \\ \text{subject to} \quad & \alpha_0{=}0,\alpha_K{=}\infty \\ & \alpha_{j-1}{<}\alpha_j,j{=}1,...,K\end{aligned} \tag{1}$$

Analysis: For convenience, we use θs to equivalently replace αs. With a traffic load of ρ, the average time in the lth queue, T_l, can be expressed as: $T_l{=}\frac{\theta_l\rho}{1-\rho F(\alpha_{l-1})}$ (assuming M/M/1 queues[3]). Since $\sum_{m=1}^{l}\theta_m{=}\sum_{m=l}^{K}F(\alpha_m){-}F(\alpha_{m-1})$, we can re-express the objective as: $\mathcal{T}{=}\sum_{l=1}^{K}T_l(1{-}F(\alpha_{l-1}))$. $\sum_{l=1}^{K}\theta_m{=}1$.

Since $\sum_{l=1}^{K}\theta_m{=}1$, we can finally transform the problem as:

$$\max_{\{\theta_l\}} \quad \mathcal{T}''{=}\sum_{l=1}^{K-1}\frac{\theta_l}{1{-}\rho(\sum_{m=1}^{l-1}\theta_m)}{+}\frac{1{-}\sum_{m=1}^{K-1}\theta_m}{1{-}\rho\sum_{m=1}^{K-1}\theta_m} \tag{2}$$

which is a Sum-of-Linear-Ratios (SoLR) problem [32], a well-known class of fractional programming problems. The only constraint is that $\theta_l{\ge}0,\forall l$. This formulation is interesting because the upperbound of average FCT is independent of the flow distribution $F(x)$, and only concerns the θs, which represent the percentages of traffic in different queues. Thus $F(x)$ is needed only when we calculate the thresholds, so we can first obtain the optimal set of θs for all links, and then derive the priority thresholds based on $F(x)$.

Solution method: Generally, the SoLR problem is $\mathcal{NP}$-hard [17], and the difficulty in solving this class of problems lies in the lack of useful properties to exploit. For our problem, however, we find a set of properties that can lead to a closed-form analytical solution to Problem 2. We describe the derivation procedure as follows:

Consider the terms in the objective. Since the traffic load $\rho{\le}1$, we have $\rho(\sum_{m=1}^{l-1}\theta_m){\le}\sum_{m=1}^{l-1}\theta_m$. Also, $\theta_l{+}\sum_{m=1}^{l-1}\theta_m{=}\sum_{m=1}^{l}\theta_m{\le}1$. Thus we have:

$$\theta_l{\le}\sum_{m=l}^{K}\theta_m{=}1{-}\sum_{m=1}^{l-1}\theta_m{\le}1{-}\rho(\sum_{m=1}^{l-1}\theta_m) \tag{3}$$

The property to exploit is as follows: each term in the summation, $\theta_l/(1{-}\rho\sum_{m=1}^{l-1}\theta_m)$, is no larger than 1, and to maximize the summation, we should make the numerator and denominator as close as possible, so that the ratio is close to 1.

Consider the first two portions, θ_1 and θ_2. By making the numerator and denominator equal, we have:

$$\theta_2{=}1{-}\rho\theta_1 \tag{4}$$

We can obtain the expression of the third portion and

[3] We use M/M/1 queues to simplify the analysis and obtain a closed-form solution. Similar derivation can also be conducted using M/G/1 queues [13], but the solution is very complicated.

all the portions after it iteratively:

$$\theta_3{=}1{-}\rho(\theta_2{+}\theta_1){=}1{-}\rho(1{-}(1{-}\rho)\theta_1) \qquad (5)$$

In this way, by the formula $\theta_l{=}1{-}\rho(\sum_{m=1}^{l-1}\theta_m)$, each portion θ_l can be represented by an expression of θ_1 iteratively. In addition, by the constraint $\sum_{l=1}^{K}\theta_l{=}1$, we can obtain the analytical expressions of all θs, which represents the percentages of the traffic in different priority queues on the link. Given traffic load θs and flow size distribution $F(\cdot)$, we can express the thresholds in the unit of bytes, and this representation is implemented at the end host.

4 Implementation and Testbed Setup

4.1 PIAS Implementation

We have implemented a prototype of PIAS. We now describe each component of our prototype in detail.

4.1.1 Packet Tagging

The packet tagging module is responsible for maintaining per-flow state and marking packets with priority at the end hosts. We implement it as a kernel module in Linux. The packet tagging module resides between the TCP/IP stack and Linux TC, which consists of three components: a NETFILTER [6] hook, a hash based flow table, and a packet modifier.

The operations are as follows: 1) the NETFILTER hook intercepts all outgoing packets using the LOCAL_OUT hook and directs them to the flow table. 2) Each flow in the flow table is identified by the 5-tuple: src/dst IPs, src/dst ports and protocol. When a packet comes in, we identify the flow it belongs to (or create a new entry) and increment the amount of bytes sent. 3) Based on the flow information, the packet modifier sets the packet's priority by modifying the DSCP field in the IP header to the corresponding value.

Offloading techniques like large segmentation offloading (LSO) may degrade the accuracy of packet tagging. With LSO, the packet tagging module may not be able to set the right DSCP value for each individual MTU-sized packet within a large segment. To quantify this, we sample more than 230,000 TCP segments with payload data in our 1G testbed and find that the average segment size is only 7,220 Bytes. This has little impact on packet tagging. We attribute this to the small window size in DCN environment which has small bandwidth-delay product and large number of concurrent connections. We expect that the final implementation solution for packet tagging should be in NIC hardware to permanently avoid this interference.

To quantify system overhead introduced by the PIAS packet tagging module, we installed it on a Dell PowerEdge R320 server with an Intel 82599EB 10GbE NIC and measured CPU usage. LSO is enabled in this experiment. We started 8 TCP long flows and achieved ~9.4Gbps goodput. The extra CPU usage introduced by PIAS is $< 1\%$ compared with the case where the PIAS packet tagging module is not enabled.

4.1.2 Switch Configuration

We enforce strict priority queues at the switches and classify packets based on the DSCP field. Similar to previous work [11, 35], we use ECN marking based on the instant queue lengths with a single marking threshold. In addition to the switch queueing delay in the network, sender NIC also introduces latency because it is actually the first contention point of the fabric [13, 24]. Hardware and software queues at the end hosts can introduce large queueing delay, which might severely degrade the application performance [23, 36]. To solve this problem, our software solution hooks into the TX datapath at POST_ROUTING and rate-limits outgoing traffic at the line rate. Then, we perform ECN marking and priority queueing at the end host as well as the switches.

Per-queue vs per-port ECN marking: We observe that some of today's commodity switching chips offer multiple ways to configure ECN marking when configured to use multiple queues per port. For example, our Broadcom BCM#56538-based switch allows us to enable either per-queue ECN marking or per-port ECN marking. In per-queue ECN marking, each queue has its own marking threshold and performs ECN marking independently to other queues. In per-port ECN marking, each port is assigned a single marking threshold and marks packets when the sum of all queue sizes belonging to the port exceeds the marking threshold.

Per-queue ECN is widely used in many DCN transport protocols [11, 26, 33], however, we find it has limitations when supporting multiple queues. Each queue requires a moderate ECN marking threshold h to fully utilize the link independently (e.g., $h{=}20$ packets for 1G and 65 packets for 10G in DCTCP [11]). Thus, supporting multiple queues may require the shared memory be at least multiple times (e.g., 8) the marking threshold, which is not affordable for most shallow buffered commodity switches. For example, our Pronto-3295 switch has 4MB ($\approx$2667 packets) memory shared by 384 queues (48x 1G ports with 8 queues per port). If we set $h{=}20$ packets as suggested above, we need over 11MB memory in the worst case, otherwise when the traffic is bursty, the shallow buffer may overflow before ECN takes effect.

Per-port ECN, to the best of our knowledge, has rarely been exploited in recent DCN transport designs. Although per-port ECN marking cannot provide ideal isolation among queues as per-queue ECN marking (Figure 12) , it can provide much better burst tolerance and support a larger number of queues in shallow buffered

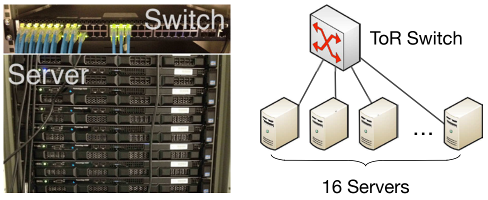


Figure 4: Testbed Topology

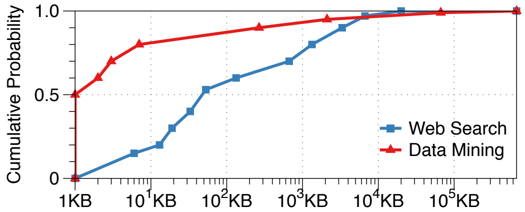


Figure 5: Traffic distributions used for evaluation.

switches. Moreover, per-port ECN marking can potentially mitigate the starvation problem. It can push back high priority flows when many packets of low priority flows get queued in the switch. Therefore, we use per-port ECN marking.

4.1.3 Rate Control

We use the open source DCTCP patch [4] for Linux 2.6.38.3. We further observe an undesirable interaction between the open-source DCTCP implementation and our switch. The DCTCP implementation does not set the ECN-capable (ECT) codepoint on TCP SYN packets and retransmitted packets, following the ECN standard [31]. However, our switch drops any non-ECT packets from ECN-enabled queues, when the instant queue length is larger than the ECN marking threshold. This problem severely degrades the TCP performance [35]. To address this problem, we set ECT on every TCP packet at the packet modifier.

4.2 Testbed Setup

We built a small testbed that consists of 16 servers connected to a Pronto 3295 48-port Gigabit Ethernet switch with 4MB shared memory, as shown in Figure 4. Our switch supports ECN and strict priority queuing with at most 8 class of service queues [1]. Each server is a Dell PowerEdge R320 with a 4-core Intel E5-1410 2.8GHz CPU, 8G memory, a 500GB hard disk, and a Broadcom BCM5719 NetXtreme Gigabit Ethernet NIC. Each server runs Debian 6.0-64bit with Linux 2.6.38.3 kernel. By default, advanced NIC offload mechanisms are enabled to reduce the CPU overhead. The base round-trip time (RTT) of our testbed is around $100us$.

In addition, we have also built a smaller 10G testbed for measuring the end host queueing delay in the high speed network. We connect three servers to the same switch (Pronto 3295 has four 10GbE ports). Each server is equipped with an Intel 82599EB 10GbE NIC.

5 Evaluation

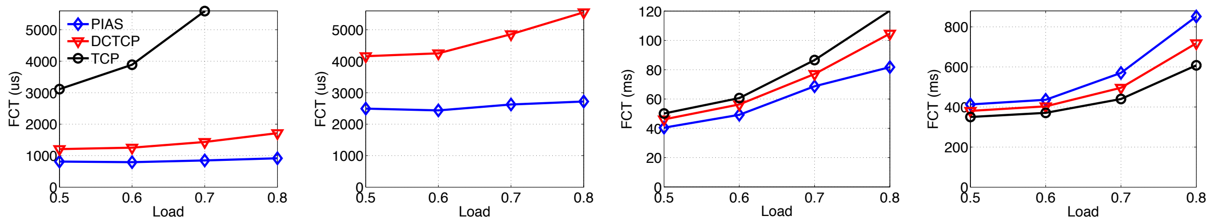
We evaluate PIAS using a combination of testbed experiments and large-scale ns-2 simulations. Our evaluation centers around four key questions:

- **How does PIAS perform in practice?** Using realistic workloads in our testbed experiments, we show that PIAS reduces the average FCT of short flows by ∼37-47% with the web search workload [11] and ∼30-45% with the data mining workload [21] compared to DCTCP. In an application benchmark with Memcached [7], we show that PIAS achieves ∼28-30% lower average query completion time than DCTCP.
- **How effective are individual design components of PIAS**, and how sensitive is PIAS to parameter settings? We show that PIAS achieves reasonable performance even with two queues. We also demonstrate that ECN is effective in mitigating the harmful effect of a mismatch between the demotion thresholds and traffic, but PIAS performs the best with the optimal threshold setting.
- **Does PIAS work well even in large datacenters?** Using large-scale ns-2 simulations, we show that PIAS scales to multi-hop topologies and performs best among all information-agnostic schemes (DCTCP [11], L2DCT [27], and LAS [30]). PIAS shows a 4.9% performance (measured in average FCT) gap from pFabric [13], an idealized information-aware scheme, for short flows in the data mining workload.
- **How robust is the PIAS threshold setting?** With the same set of thresholds, we experiment with both extremely-biased and realistic traffic patterns to show the robustness of the PIAS threshold setting. We also demonstrate the optimality region and close-to-optimality region is large for PIAS analytically. (For space limitations, please see the details in our technical report [15]).

5.1 Testbed Experiments

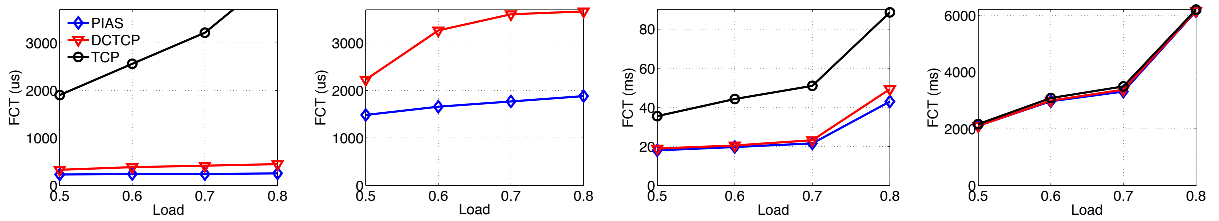
Setting: PIAS uses 8 priority queues by default and enable per-port ECN marking as discussed in Section 4. We set the ECN marking threshold to be 30KB as DCTCP [11] recommends. Our MLFQ demotion thresholds are derived as described in Section 3.

We use two realistic workloads, a web search workload [11] and a data mining workload [21] from production datacenters. Their overall flow size distributions are shown in Figure 5. We also evaluate PIAS using an application benchmark with Memcached [7].



(a) (0,100KB]: Avg (b) (0,100KB]: 99th Percentile (c) (100KB,10MB]: Avg (d) (10MB,∞): Avg

Figure 6: Web search workload: FCT across different flow sizes. TCP's performance is outside the plotted range of (b)



(a) (0,100KB]: Avg (b) (0,100KB]: 99th Percentile (c) (100KB,10MB]: Avg (d) (10MB,∞): Avg

Figure 7: Data mining workload: FCT across different flow sizes. TCP's performance is outside the plotted range of (b)

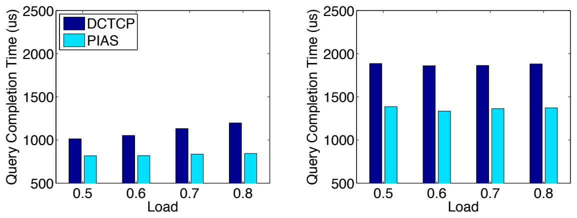
Results with realistic workloads: For this experiment, we developed a client/server model to generate dynamic traffic according to realistic workloads and measure the FCT on application layer. The client application, running on 1 machine, periodically generates requests to the other machines to fetch data. The server applications, running on 15 other machines, respond with requested data. The requests are generated based on a Poisson process. We evaluate the performance of PIAS, DCTCP and TCP, while varying the network loads from 0.5 to 0.8. Given the average traffic load in DCNs is moderate (for exmaple, 30% [16]), a long-term load of over 80% is less likely in practice.

Figure 6 and Figure 7 show the average FCT across small (0,100KB] (a, b), medium (100KB,10MB] (c), and large (10MB,∞) (d) flows, respectively; for the web search and data mining workloads, respectively.

We make the following three observations: First, for both workloads, PIAS achieves the best performance in both the average and 99th percentile FCTs of small flows. Compared to DCTCP, PIAS reduces the average FCT of small flows by ∼37-47% for the web search workload and ∼30-45% for the data mining workload. The improvement of PIAS over DCTCP in the 99th percentile FCT of short flows is even larger: ∼40-51% for the web search workload and ∼33-48% for the data mining workload. Second, PIAS also provides the best performance in medium flows. It achieves up to 22% lower average FCT of medium flows than DCTCP in the web search workload. Third, PIAS does not severely penalize the large flows. For example, from Figure 6 (d) and Figure 7 (d), we can see that for the data mining workload PIAS is comparable or slightly better than TCP and DCTCP, while for the web search workload it is worse than DCTCP by over 10%. This is expected because PIAS prioritizes short flows over long flows and ∼60% of all bytes in the web search workload are from flows smaller than 10MB. Note that this performance gap would not affect the overall average FCT since datacenter workloads are dominated by small and medium flows (e.g., only ∼3% flows are larger than 10MB in the web search workload). In fact, PIAS achieves ∼9% and ∼17% lower overall average FCT than DCTCP and TCP at 0.8 load in the web search workload.

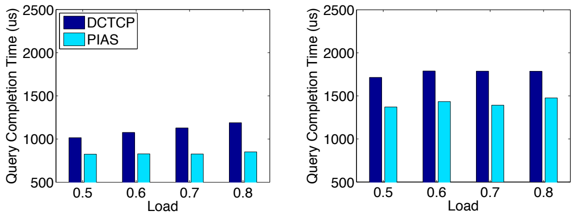
Results with the Memcached application: To assess how PIAS improves the performance of latency-sensitive applications, we build a Memcached [7] cluster with 16 machines. One machine is used as a client and the other 15 are used as servers to emulate a partition/aggregate soft-real-time service [11, 34]. We pre-populate server instances with 4B-key, 1KB-value pairs. The client sends a GET query to all 15 servers and each server responds with a 1KB value. A query is completed only when the client receives all the responses from the servers. We measure the query completion time as the application performance metric. Since a 1KB response can be transmitted within one RTT, the query completion time is mainly determined by the tail queueing delay. The base query completion time is around 650us in our testbed. We also generate background traffic, a mix of mice flows and elephant flows following the distribution of the web search workload [11]. We use queries per second, or qps, to denote the application load. We vary the load of the background traffic from 0.5 to 0.8 and compare the performance of PIAS with that of DCTCP.

Figure 8 and Figure 9 show the results of the query completion time at 20 and 40 qps loads respectively. S-



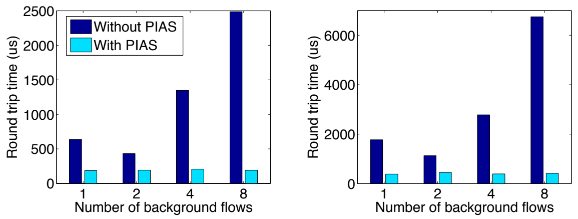
(a) Mean (b) 99th Percentile

Figure 8: Query completion time at 20 qps



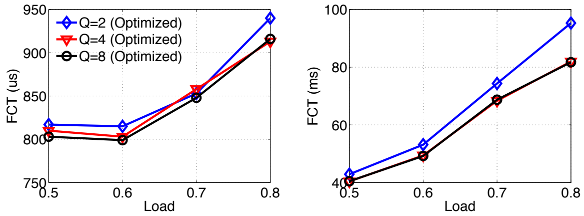
(a) Mean (b) 99th Percentile

Figure 9: Query completion time at 40 qps



(a) Mean (b) 99th Percentile

Figure 10: RTT with background flows



(a) (0,100KB]: Avg (b) (100KB,10MB]: Avg

Figure 11: Web search workload with different queues

ince we enable both dynamic buffer management and ECN on the switch, none of queries suffers from TCP timeout. With the increase in background traffic load, the average query completion time of DCTCP also increases (1016–1189us at 40qps and 1014–1198us at 20qps). By contrast, PIAS maintains a relatively stable performance. At 0.8 load, PIAS can achieve ~28-30% lower average query completion times than those of DCTCP. Moreover, PIAS also reduces the 99th percentile query completion time by ~20-27%. In summary, PIAS can effectively improve the performance of the Memcached application by reducing the queueing delay of short flows.

End host queueing delay: The above experiments mainly focus on network switching nodes. PIAS extends its switch design to the end hosts as the sender's NIC is actually the first contention point of the fabric [13, 24].

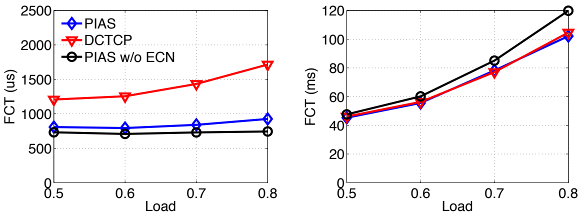
To quantify this, we conduct an experiment in our 10G setting with three servers (one sender and two receivers). We start several (1 to 8) long-lived TCP flows from the sender to a receiver. Then we measure RTT from the sender to the other receiver by sending ICMP `ping` packets. Without PIAS, `ping` packets could experience up to 6748us queueing delay with 8 background flows. Then we deploy a 2-queue PIAS end host scheduling module (as described in §4.1.2) with a threshold of 100KB. Each ICMP packet is identified as a new flow by PIAS. We measure the RTTs with PIAS and compare them with the results without PIAS in Figure 10. In general, PIAS can significantly reduce the average RTT to ~200us and ensure that the 99th percentile RTT is smaller than 450us. Note that the PIAS scheduling module does not affect network utilization and large flows still maintain more than 9Gbps goodput during the experiment. Since we enable LSO to reduce CPU overhead, it is difficult for us to achieve fine-grained transmission control and some delay may still exist in NIC's transmission queues. We believe there is still room to improve by offloading the scheduling to NIC hardware [29].

5.2 PIAS Deep Dive

In this section, we conduct a series of targeted experiments to answer the following three questions:

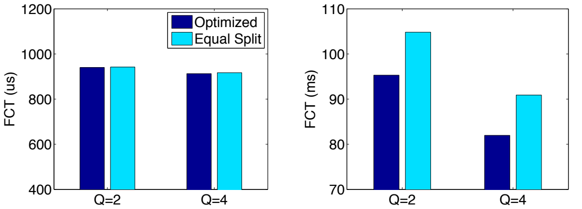
- **How sensitive is PIAS to the number of queues available?** Network operators may reserve some queues for other usage while some commodity switches [3] only support 2 priority queues. We find that, even with only 2 queues, PIAS still effectively reduces the FCT of short flows. However, in general, more queues can further improve PIAS's overall performance.
- **How effective is ECN in mitigating the mismatch?** ECN is integrated into PIAS to mitigate the mismatch between the demotion thresholds and traffic distribution. In an extreme mismatch scenario, we find that without ECN, PIAS's performance suffers with medium flows and is worse than DCTCP. However, with ECN, PIAS effectively mitigates this problem, and is better than, or at least comparable to DCTCP.
- **What is the effect of the optimal demotion thresholds?** Compared to PIAS with thresholds derived from simple heuristics, PIAS with the optimal demotion thresholds achieves up to ~10% improvement in medium flows, which improves the overall performance.

Impact of number of queues: In general, the more queues we use, the better we can segregate different flows, thus improving overall performance. For this



(a) (0,100KB]: Avg (b) (100KB,10MB]: Avg

Figure 12: Web search workload with mismatch thresholds derived from data mining workload

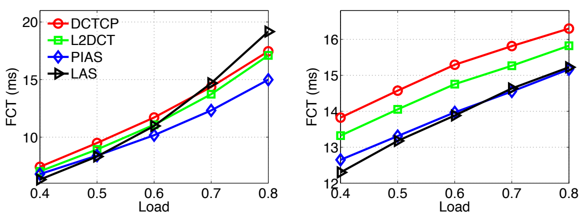


(a) (0,100KB]: Avg (b) (100KB,10MB]: Avg

Figure 13: Web search workload with different thresholds

experiment, we generate traffic using the web search workload and do the evaluation with 2, 4 and 8 priority queues. The results are shown in Figure 11. We observe that three schemes achieve the similar average FCT of short flows. Even at 0.8 load, the FCT of 2 queues is within ∼2.5% of that of 8 queues. As expected, the average FCT of medium flows improves with the increasing number of queues. For example, PIAS with 4 queues and 8 queues provide similar performance, but improve the FCT by 14.3% compared to 2 queues. The takeaway is that PIAS can effectively reduce FCT of short flows even with 2 queues and more queues can further improve PIAS's overall performance.

Effect of ECN under thresholds–traffic mismatch: We evaluate the performance of the web search workload while using the optimal demotion thresholds derived from the data mining workload. We compare PIAS, PIAS without ECN, and DCTCP. Figure 12 shows the results of the average FCT of short and medium flows. Both PIAS and PIAS without ECN greatly outperforms DCTCP in short flows. PIAS without ECN is even slightly better than PIAS. That is because, when using per-port ECN, packets in a high priority queue may get marked due to buffer occupation in a low priority queue. However, PIAS without ECN shows the worst performance in medium flows while being obviously worse than D-CTCP. This is because, due to the mismatch between demoting thresholds and traffic distribution, medium flows and large flows coexist in the lower priority queues. Without ECN, packets from medium flows would experience queueing delay behind large flows. With EC-N, PIAS effectively mitigates this side-effect by keeping low buffer occupation as explained in §3.2.2.



(a) Web search workload (b) Data mining workload

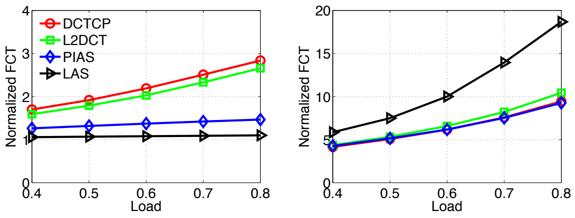
Figure 14: Overall average flow completion time

Impact of demotion thresholds: To explore the effectiveness of our optimal demotion threshold setting, we compare the optimized PIAS with the PIAS using thresholds derived from the equal split heuristic as [13]. More specifically, given a flow size distribution and N queues, we set the first threshold to the size of $100/N$th percentile flow, the second threshold to the size of $200/N$th percentile flow, and so on. We run the web search workload at 0.8 load and summarize results in the Figure 13. We test PIAS with 2 and 4 queues. We observe that there is an obvious improvement in the average FCT of medium flows with the optimized thresholds. Specifically, PIAS (4-queue) with the optimized thresholds can achieve ∼10% lower FCT for medium flows than that of equal split, and a 9% improvement for the 2-queue PIAS. This partially validates the effectiveness of our optimal threshold setting. We further conduct deep analysis on this in [15].

5.3 Large-scale NS-2 Simulations

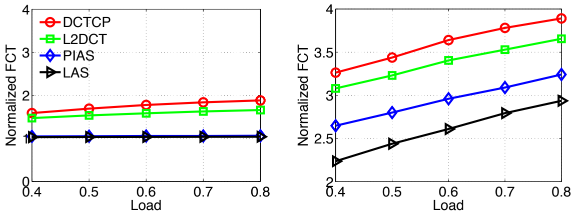
We use ns-2 [10] simulations to answer three questions.

- **How does PIAS perform compared to information-agnostic schemes?** PIAS outperforms DCTCP [11] and L2DCT [27] in general, and significantly improves their average FCTs for short flows by 50% and 40% respectively. Furthermore, PIAS is close to LAS for short flows and greatly outperforms LAS for long flows, reducing its average FCT by 50% in the web search workload.
- **How does PIAS perform compared to information-aware schemes?** As a practical information-agonistic scheme, PIAS can also deliver comparable performance to a clean-slate information-aware design, p-Fabric [13], in certain scenarios. For example, it only has a 4.9% gap to pFabric for short flows in the data mining workload.
- **How does PIAS perform in the oversubscribed network?** In a 3:1 oversubscribed topology with ECMP load balancing, PIAS still delivers very good performance. Compared to DCTCP, the average FCT for the short flows with PIAS is ∼26% lower in the web search workload.



(a) (0,100KB]: Avg (b) (10MB,∞): Avg

Figure 15: Web search workload: Normalized FCT



(a) (0,100KB]: Avg (b) (10MB,∞): Avg

Figure 16: Data mining workload: Normalized FCT

Setting: We use a leaf-spine topology with 9 leaf (ToR) switches to 4 spine (Core) switches. Each leaf switch has 16 10Gbps downlinks (144 hosts) and 4 40Gbps uplinks to the spine, forming a non-oversubscribed network. The base end-to-end round-trip time across the spine (4 hops) is 85.2μs. We use packet spraying [19] for load balancing and disable dupACKs to avoid packet reordering. Again, we use the web search and data mining workloads as above.

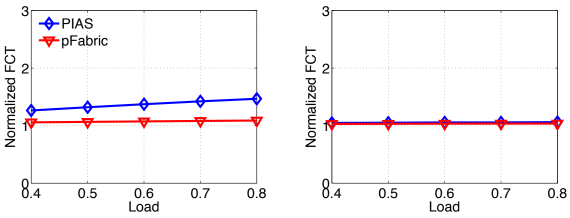
5.3.1 Comparison with Information-agnostic Schemes

We mainly compare PIAS with three other information-agnostic schemes: DCTCP, L2DCT [27] and LAS [30].

Overall performance: Figure 14 shows the average FCT of information-agnostic schemes under different workloads and load levels. From the figure, we see that PIAS performs well overall. First, PIAS has an obvious advantage over DCTCP and L2DCT in all cases. Second, PIAS is close to LAS in the data mining workload, and significantly outperforms LAS by 28% (at 0.8 load) in the web search workload. This is because PIAS effectively mitigates the starvation between long flows unlike LAS. In the data mining workload, there are not so many large flows on the same link concurrently. As a result, LAS does not suffer from starvation as significantly.

Breakdown by flow size: We now breakdown the average FCT across different flow sizes, (0, 100KB] and (10MB, ∞) (Figure 15 and 16). We normalize each flow's actual FCT to the best possible value it can achieve in the idle fabric.

For short flows in (0,100KB], we find that PIAS significantly outperforms both DCTCP and L2DCT, improving the average FCT by up to 50% and 40% respectively. This is because DCTCP and L2DCT use reactive rate control at the end hosts, which is not as effective as PIAS for in-network flow scheduling. We further observe that PIAS achieves similar performance as LAS for short flows. PIAS only performs slightly worse than LAS in the web search workload when there is a packet drop or an explicit congestion notification.



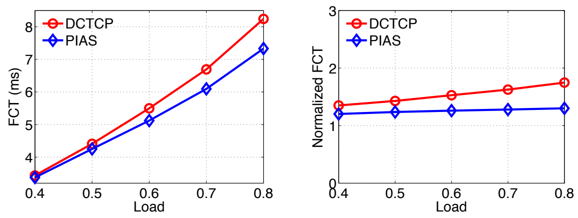
(a) Web search workload (b) Data mining workload

Figure 17: Average normalized FCT for (0,100KB]

For long flows in (10MB,∞), we find that PIAS is slightly worse than LAS in data mining workload, but performs significantly better in the web search workload (50% reduction in FCT at 0.8 load). This is because, in the web search workload, it is common that multiple large flows are present in the same link. In such scenarios, LAS always stops older flows to send new flows. Since large flows usually take a very long time to complete, it causes a serious starvation problem. However, with PIAS, large flows receive their fair sharing in the lowest priority queue, which mitigates this problem. Furthermore, PIAS performs similarly to DCTCP under the web search workload and achieves 17% lower FCT in the data mining workload.

5.3.2 Comparison with Ideal Information-aware Schemes

We compare PIAS to an ideal information-aware approach for DCN transport, pFabric [13], on small flows of the two workloads. We note that the most recent work PASE [26] can achieve better performance than pFabric in particular scenarios (e.g., very high load and single rack). However in our topology setting with realistic workloads, pFabric is better than PASE and PDQ [22], and achieves near-optimal performance. Thus, we directly compare PIAS with pFabric. The result is shown in Figure 17. In general, PIAS delivers comparable average FCT for short flows as pFabric, particularly within 4.9% in the data mining workload. We find that the gap between PIAS and pFabric is smaller in the data mining workload than that in the web search workload. This is mainly due to the fact that the data mining workload is more skewed than the web search workload. Around 82% flows in the data mining are smaller than 100KB, while only 54% of flows in the web search are small-



(a) Overall: Avg (b) (0,100KB]: Avg

Figure 18: Web search workload on a 3:1 oversubscribed topology with ECMP load balancing.

er than 100KB. For the web search workload, it is more likely that large flows coexist with short flows in the high priority queues temporarily, increasing the queueing delay for short flows. pFabric, by assuming prior knowledge of flow sizes, is immune to such problem.

5.3.3 Performance in oversubscribed network

Finally, we evaluate PIAS on a 3:1 oversubscribed network with ECMP load balancing. In this topology, there are 3 leaf switches and 4 spine switches. Each leaf switch is connected to 48 hosts with 10Gbps links and 4 spine switches with 40Gbps links. Given that the source and destination of each flow is generated randomly, one-third of traffic is intra-ToR and the rest is inter-ToR traffic. Hence, the load at the fabric's core is twice the load at the edge. We repeat the web search workload and compare PIAS with DCTCP. Figure 18 gives the results. Note that the load in the figure is at the fabric's core. Compared to DCTCP, PIAS achieves up to ∼26% and ∼11% lower average FCT for short flows and all the flows respectively.

6 Related Work

We classify previous work on minimizing FCT in DCNs into two categories: information-agnostic solutions (e.g., [11, 12, 27]) and information-aware solutions (e.g., [13, 22, 26]).

Information-agnostic solutions [11, 12, 27] generally improve the FCT for short flows by keeping low queue occupancy. For example, DCTCP [11] tries to keep the fabric queues small by employing an ECN-based adaptive congestion control algorithm to throttle long elephant flows. L2DCT [27] adds bytes sent information to DCTCP [11]. HULL [12] further improves the latency of DCTCP by trading network bandwidth. In summary, these solutions mainly perform end-host based rate control which is ineffective for flow scheduling. By contrast, PIAS leverages in-network priority queues to emulate SJF for flow scheduling, which is more efficient in terms of FCT minimization.

Information-aware solutions [13, 22, 26] attempt to approximate ideal Shortest Remaining Processing Time (SRPT) scheduling. For example, PDQ [22] employs switch arbitration and uses explicit rate control for flow scheduling. pFabric [13] decouples flow scheduling from rate control and achieves near-optimal FCT with decentralized in-network prioritization. PASE [26] synthesizes the strengths of previous solutions to provide good performance. In general, these solutions can potentially provide ideal performance, but they require non-trivial modifications on switch hardware or a complex control plane for arbitration. By contrast, PIAS does not touch the switch hardware or require any arbitration in the control plan, while still minimizing FCT.

There are also some other efforts [18, 33, 34] targeting at meeting flow deadlines. D3 [34] assigns rates to flows according to their sizes and deadlines explicitly, whereas D2TCP [33] and MCP [18] add deadline-awareness to ECN-based congestion window adjustment implicitly. They all require prior knowledge of flow information and do not directly minimize FCT, unlike PIAS.

7 Conclusion

Through PIAS, we leverage existing commodity switches in DCNs to minimize the average FCT for flows, especially the smaller ones, without assuming any prior knowledge of flow sizes. We have implemented a PIAS prototype using all commodity hardware and evaluated PIAS through a series of small-scale testbed experiments as well as large-scale packet-level ns-2 simulations. Both our implementation and evaluation results show that PIAS is a viable solution that achieves all our design goals.

Acknowledgements

This work was supported by the Hong Kong RGC ECS 26200014, the China 973 Program under Grant No. 2014CB340303, the ICT R&D program of MSIP/IITP, Republic of Korea [14-911-05-001], and the Basic Science Research Program of NRF funded by MSIP, Rep. of Korea (2013R1A1A1076024). We thank Mohammad Alizadeh for sharing codes of pFabric, Haitao Wu for insightful discussions, our shepherd Dushyanth Narayanan and the anonymous NSDI reviewers for their constructive comments.

References

[1] http://www.pica8.com/documents/pica8-datasheet-picos.pdf.

[2] "Apache Storm," https://storm.incubator.apache.org/.

[3] "Cisco Nexus 5500 Series NX-OS Quality of Service Configuration Guide," http://www.cisco.com/c/en/us/td/docs/switches/datacenter/nexus5500/sw/qos/7x/b_5500_QoS_Config_7x.pdf.

[4] "DCTCP Patch," http://simula.stanford.edu/~alizade/Site/DCTCP.html.

[5] "Hadoop," http://hadoop.apache.org/.

[6] "Linux netfilter," http://www.netfilter.org.

[7] "Memcached," http://memcached.org/.

[8] "Microsoft SQL Server," http://www.microsoft.com/en-us/server-cloud/products/sql-server/.

[9] "OpenStack Object Storage," http://docs.openstack.org/api/openstack-object-storage/1.0/content/chunked-transfer-encoding.html.

[10] "The Network Simulator NS-2," http://www.isi.edu/nsnam/ns/.

[11] 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*.

[12] M. Alizadeh, A. Kabbani, T. Edsall, B. Prabhakar, A. Vahdat, and M. Yasuda, "Less is more: trading a little bandwidth for ultra-low latency in the data center," in *NSDI 2012*.

[13] M. Alizadeh, S. Yang, M. Sharif, S. Katti, N. McKeown, B. Prabhakar, and S. Shenker, "pFabric: Minimal near-optimal datacenter transport," in *SIGCOMM 2013*.

[14] W. Bai, L. Chen, K. Chen, D. Han, C. Tian, and W. Sun, "PIAS: Practical Information-Agnostic Flow Scheduling for Datacenter Networks," in *HotNets 2014*.

[15] W. Bai, L. Chen, K. Chen, D. Han, C. Tian, and H. Wang, "Information-Agnostic Flow Scheduling in Commodity Data Centers," http://sing.cse.ust.hk/papers/pias-tr.pdf, Technical Report HKUST-CS15-01, 2015.

[16] T. Benson, A. Akella, and D. A. Maltz, "Network Traffic Characteristics of Data Centers in the Wild," in *IMC 2010*.

[17] J. G. Carlsson and J. Shi, "A linear relaxation algorithm for solving the sum-of-linear-ratios problem with lower dimension," *Operations Research Letters*, vol. 41, no. 4, pp. 381–389, 2013.

[18] L. Chen, S. Hu, K. Chen, H. Wu, and D. Tsang, "Towards Minimal-Delay Deadline-Driven Data Center TCP," in *HotNets 2013*.

[19] A. Dixit, P. Prakash, Y. C. Hu, and R. R. Kompella, "On the impact of packet spraying in data center networks," in *INFOCOM 2013*.

[20] R. Fielding, J. Gettys, J. Mogul, H. Frystyk, L. Masinter, P. Leach, and T. Berners-Lee, "Hypertext transfer protocol–HTTP/1.1, 1999," *RFC2616*, 2006.

[21] A. Greenberg, J. R. Hamilton, 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*.

[22] C.-Y. Hong, M. Caesar, and P. Godfrey, "Finishing flows quickly with preemptive scheduling," in *SIGCOMM 2012*.

[23] K. Jang, J. Sherry, H. Ballani, and T. Moncaster, "Silo: Predictable Message Completion Time in the Cloud," Tech. Rep. MSR-TR-2013-95, 2013.

[24] V. Jeyakumar, M. Alizadeh, D. Mazieres, B. Prabhakar, C. Kim, and A. Greenberg, "EyeQ: practical network performance isolation at the edge," in *NSDI 2013*.

[25] B. Kalyanasundaram and K. R. Pruhs, "Minimizing flow time nonclairvoyantly," *Journal of the ACM (JACM)*, vol. 50, no. 4, pp. 551–567, 2003.

[26] A. Munir, G. Baig, S. M. Irteza, I. A. Qazi, A. Liu, and F. Dogar, "Friends, not Foes - Synthesizing Existing Transport Strategies for Data Center Networks," in *SIGCOMM 2014*.

[27] A. Munir, I. A. Qazi, Z. A. Uzmi, A. Mushtaq, S. N. Ismail, M. S. Iqbal, and B. Khan, "Minimizing flow completion times in data centers," in *INFOCOM 2013*.

[28] Y. Peng, K. Chen, G. Wang, W. Bai, Z. Ma, and L. Gu, "HadoopWatch: A First Step Towards Comprehensive Traffic Forecasting in Cloud Computing," in *INFOCOM 2014*.

[29] S. Radhakrishnan, Y. Geng, V. Jeyakumar, A. Kabbani, G. Porter, and A. Vahdat, "SENIC: scalable NIC for end-host rate limiting," in *NSDI 2014*.

[30] I. A. Rai, G. Urvoy-Keller, M. K. Vernon, and E. W. Biersack, "Performance Analysis of LAS-based Scheduling Disciplines in a Packet Switched Network," in *SIGMETRICS 2004*.

[31] K. Ramakrishnan, S. Floyd, and D. Black, "RFC 3168: The addition of explicit congestion notification (ECN) to IP," 2001.

[32] S. Schaible and J. Shi, "Fractional programming: the sum-of-ratios case," *Optimization Methods and Software*, vol. 18, no. 2, pp. 219–229, 2003.

[33] B. Vamanan, J. Hasan, and T. Vijaykumar, "Deadline-aware datacenter tcp (d2tcp)," in *SIGCOMM 2012*.

[34] C. Wilson, H. Ballani, T. Karagiannis, and A. Rowtron, "Better never than late: Meeting deadlines in datacenter networks," in *SIGCOMM 2011*.

[35] H. Wu, J. Ju, G. Lu, C. Guo, Y. Xiong, and Y. Zhang, "Tuning ECN for data center networks," in *CoNEXT 2012*.

[36] Y. Xu, M. Bailey, B. Noble, and F. Jahanian, "Small is Better: Avoiding Latency Traps in Virtualized Data Centers," in *SOCC 2013*.

[37] M. Yu, A. G. Greenberg, D. A. Maltz, J. Rexford, L. Yuan, S. Kandula, and C. Kim, "Profiling Network Performance for Multi-tier Data Center Applications." in *NSDI 2011*.

A General Approach to Network Configuration Analysis

Ari Fogel   Stanley Fung   Luis Pedrosa   Meg Walraed-Sullivan
Ramesh Govindan   Ratul Mahajan   Todd Millstein

University of California, Los Angeles   University of Southern California   Microsoft Research

Abstract— We present an approach to detect network configuration errors, which combines the benefits of two prior approaches. Like prior techniques that analyze configuration files, our approach can find errors proactively, before the configuration is applied, and answer "what if" questions. Like prior techniques that analyze data-plane snapshots, our approach can check a broad range of forwarding properties and produce actual packets that violate checked properties. We accomplish this combination by faithfully deriving and then analyzing the data plane that would emerge from the configuration. Our derivation of the data plane is fully declarative, employing a set of logical relations that represent the control plane, the data plane, and their relationship. Operators can query these relations to understand identified errors and their provenance. We use our approach to analyze two large university networks with qualitatively different routing designs and find many misconfigurations in each. Operators have confirmed the majority of these as errors and have fixed their configurations accordingly.

1 Introduction

Configuring networks is arduous because policy requirements (for resource management, access control, etc.) can be complex and configuration languages are low-level. Consequently, configuration errors that compromise availability, security, and performance are common [7, 21, 36]. In a recent incident, for example, a misconfiguration led to a nation-wide outage that impacted all customers of Time Warner for over an hour [3].

Prior approaches Researchers have developed two main approaches to detect network configuration errors. The first approach directly analyzes network configuration files [2, 5, 7, 24, 25, 28, 34]. Such a *static* analysis can flag errors proactively, before a new configuration is applied to the network, and it can naturally answer "what if" questions with respect to different environments (i.e., failures and route announcement from neighbors). However, configurations of real networks are complex, with many interacting aspects (e.g., BGP, OSPF, ACLs, VLANs, static routing, route redistribution); existing configuration analysis tools handle this complexity by developing customized models for specific aspects of the configuration or specific correctness properties. For instance, rcc [7] produces a normalized representation of configuration that lets it check a range of properties that correspond to common errors (e.g., "route validity" of BGP, whether OSPF adjacencies are configured on both ends, and that there are no duplicate router identifiers). Similarly, FIREMAN [34] produces a "rule graph" structure to represent each ACL and analyzes these graphs. This selective focus makes configuration analysis practical, but it also limits the scope of what can be checked. Further, because many aspects of the configuration are not analyzed, it can be difficult for operators to assess how and whether identified errors ultimately impact forwarding.

Researchers have recently proposed a second approach that can be used to detect configuration errors: analyzing the data plane snapshots (i.e., forwarding behavior) of the network [13, 14, 22, 37]. Unlike with static analysis, any configuration error that causes undesirable forwarding can be precisely detected, because the data plane reflects the combined impact of all configuration aspects. Further, because the data plane has well-understood semantics and can be efficiently encoded in various logics, a wide range of forwarding properties can be concisely expressed and scalably checked with off-the-shelf constraint solvers.

Unfortunately, analysis of data plane snapshots cannot prevent errors proactively, before undesirable forwarding occurs. Further, once a problem is flagged, the operators still need to localize the responsible snippets of configuration. This task is challenging because the relationship between configuration snippets and forwarding behavior is complex. The responsible snippet is not necessarily

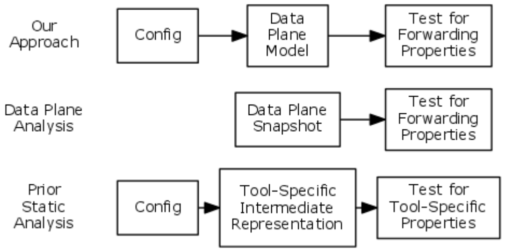

Our Approach
Config
Data Plane Model
Test for Forwarding Properties
Data Plane Analysis
Data Plane Snapshot
Test for Forwarding Properties
Prior Static Analysis
Config
Tool-Specific Intermediate Representation
Test for Tool-Specific Properties

Figure 1: *Our approach versus prior approaches.*

the most recent configuration change either; the impact of an erroneous change may only manifest long after it is introduced. For instance, the impact of erroneously configured backup paths will manifest only after a failure.

Our approach We develop a new, general approach to statically analyze network configurations that combines the strengths of the approaches above. Instead of using a customized representation, our analysis *derives the actual data plane that would emerge* given a configuration and environment. Figure 1 illustrates our approach. With it, as with prior static approaches, operators can detect errors proactively and conduct "what if" analysis across different environments. Further, as with data-plane analysis approaches, they can easily express and check a wide range of correctness properties and directly understand the impact of errors on forwarding.

Realizing our approach The principal challenge that we face is the need to derive a faithful data plane for a given configuration and environment. Our analysis must balance two competing concerns. It must be detailed and low-level in order to produce an accurate data plane, which requires us to tractably reason about all aspects of configuration and their interactions, as well as a plethora of configuration parameters and directives. At the same time, the analysis must provide a high-level view that allows operators to understand the identified errors and map them back to responsible configuration snippets.

We address this challenge in our tool, called Batfish, by implementing our analysis fully declaratively. We translate the network configuration and environment into a variant of Datalog and also use this language to express the behaviors of the various protocols being configured. Executing the resulting Datalog program produces logical relations that represent the data plane as well as relations for various key concepts in the computation, e.g., the best route to a destination as determined by a particular protocol. We use an automatic constraint solver to check properties of the resulting data plane and produce concrete packets that violate these properties. Finally, those packets are fed back into our declarative model, inducing more relational facts (e.g., the path taken, the ACL rules encountered along the way). These relations and the ones described above provide a simple ontology for understanding errors and their provenance.

Operators can query Batfish for any correctness property that can be expressed as a first-order-logic formula over the data-plane relations. However, Batfish can find errors even without operator input; by default the tool checks three novel properties related to the consistency of forwarding. Our *multipath consistency* property requires that, in the presence of multipath routing, packets of a flow are either dropped along all paths they traverse or reach the destination along all paths. Our *failure consistency* and *destination consistency* properties uncover errors that respectively limit fault tolerance and make the network vulnerable to illegitimate route announcements.

We used Batfish to find violations of these three properties in the configurations of two large university campus networks. We find many violations of each type, the majority of which the operators confirmed to be configuration errors. Because of helpful provenance information provided by Batfish, several of the errors were fixed within a day of us reporting them.

Summary We develop a new approach and a practical tool to analyze network configurations. At its heart is a high-fidelity declarative model of low-level network configurations. We believe that this model is useful beyond detecting configuration errors. For instance, researchers have proposed high-level, declarative languages to program networks [9, 18, 19, 26], but a major hurdle in adopting them is migrating a network while faithfully preserving its forwarding policies. Our model can provide a migration path. Our tool is publicly available [1] for others to use and explore various use cases.

2 Background and Motivation

This section provides background on routing in today's networks and motivates our approach.

2.1 Background

A network forwards packets through a sequence of routers and switches. The *data plane* state of each device determines how packets with a given header are handled (e.g., dropped, forwarded to a specific neighbor, or load balanced across multiple neighbors). This state is generated by the *control plane*. In today's networks, the control plane is specified through device configuration, which uses vendor-specific languages and includes aspects such as ACLs that specify packet filtering policies, static routes for IP address prefixes that are directly connected, and directives for one or more routing protocols. Configurations of all devices, combined with the current topology and dynamic information exchanged between neighboring devices, determine the current data plane.

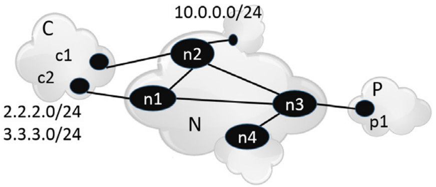
A network managed by some administrative entity is known as an *autonomous system* (AS). Within an AS, information on network topology and connected destinations is exchanged using interior gateway protocols such as OSPF [23], a protocol that computes least-cost paths. BGP [29], a protocol that accommodates policy constraints, is used across ASes. Routers announce destination IP address prefixes to which they are willing to carry traffic from a neighboring AS. Local policy determines if a received announcement is acceptable (e.g., whether the announcer can be trusted to have a path to the destination prefix) and which one among the multiple announcements for the same prefix should be selected (e.g., based on commercial relationships).

As an aside, in the SDN paradigm, which has gained significant attention of late, the control plane is specified using a control program instead of configuration. We focus on the configuration-based paradigm because it currently dominates and continues to be a cause of subtle errors. Even if SDNs become dominant, many networks will likely continue to be configuration-based, in the same way that legacy software is prevalent despite the advent of higher-level programming technologies.

2.2 Motivation

Given the complexity of network configurations, errors are common [21, 31, 36], and operators need good tools to flag potential errors. Consider network N pictured at the top of Figure 2, with two neighboring ASes. P is a large provider AS, and C is a customer AS that owns two destination prefixes. Router $n2$ is directly connected to an internal private network with prefix 10.0.0.0/24. The operators intend that this network be available to C, but not to P or other parts of N not servicing C.

The bottom of Figure 2 shows configuration snippets that implement this specification, loosely based on Cisco's IOS language. The first two lines of $n1$'s configuration specify that it runs OSPF on interfaces that connect it to $n2$ and $n3$, each with routing cost metric of 1. The next two specify that it runs BGP with $c2$ and will accept only announcements for prefixes that match the prefix list *PL_C*. Router $n2$ is similarly configured except that it also redistributes (i.e., advertises) connected networks through OSPF. Router $n3$ is configured to accept all prefix announcements from $p1$ and to redistribute into OSPF all statically configured networks. To isolate prefix 10.0.0.0/24 from nodes not on the path to C, the operator installs a static discard route with logging at $n3$ (line 6). This route is redistributed (line 4) so $n4$ need not be directly aware of this route. This setup prevents P and $n4$ (and hosts behind them) from accessing 10.0.0.0/24 and enables the operators to discover any attempts.

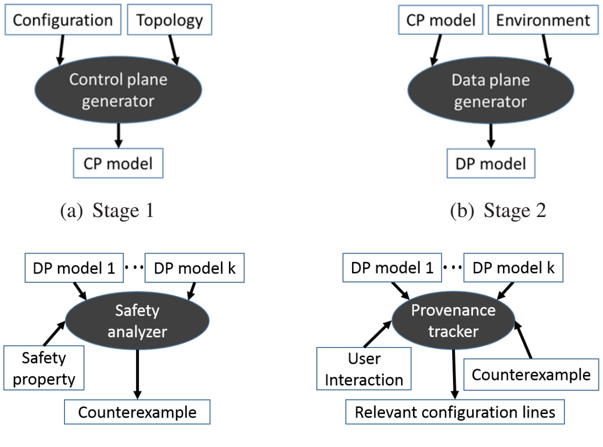


```
//----------Configuration of n1----------
1 ospf interface int1_2 metric 1
2 ospf interface int1_3 metric 1

3 prefix-list PL_C 2.2.2.0/24 3.3.3.0/24

4 bgp neighbor c2 AS C apply PL_C

//----------Configuration of n2----------
1 ospf interface int2_1 metric 1
2 ospf interface int2_3 metric 1
3 ospf-passive interface int2_5 ip 10.0.0.0/24
4 ospf redistribute connected metric 10

5 prefix-list PL_C 2.2.2.0/24

6 bgp neighbor c1 AS C apply PL_C

//----------Configuration of n3----------
1 ospf interface int3_1 metric 1
2 ospf interface int3_2 metric 1
3 ospf interface int3_4 metric 1

4 ospf redistribute static metric 10

5 bgp neighbor p1 AS P Accept ALL

6 static route 10.0.0.0/24 drop, log
```

Figure 2: *Example network configuration snippets.*

The example above is based on actual configurations of a large university network that we have analyzed using Batfish, and, despite its simplicity, it has at least two errors. The first error is that 3.3.3.0/24 is missing from the definition of *PL_C* in $n2$, and thus $n2$ will drop announcements and not provide connectivity for this prefix. This error may go unnoticed when the configuration is applied since connectivity to 3.3.3.0/24 is available through $n1$. But when $n1$, $c2$ or link $c2$-$n1$ fails, all connectivity to 3.3.3.0/24 will be lost. The end result of this error is lack of fault tolerance and poor load balancing (since link $c2$-$n1$ carries all traffic for 3.3.3.0/24).

The second error is more subtle. Because $n2$ and $n3$ redistribute connected and static networks, respectively, $n1$



(a) Stage 1

(b) Stage 2

(c) Stage 3

(d) Stage 4

Figure 3: *The four stages of* Batfish *workflow.*

will learn paths to 10.0.0.0/24 from both these neighbors, and the paths will have the same routing cost. Under these conditions, the default is multipath routing; that is, *n*1 will send packets to 10.0.0.0/24 through both neighbors. However, only packets sent through *n*2 will reach the destination since *n*3 will drop such packets. Thus, traffic sources will experience intermittent connectivity.[1]

No existing technique can find both of these errors proactively, before the buggy configuration is applied. Data plane analysis can detect reachability issues but it will not find the first error until a failure occurs that breaks reachability to 3.3.3.0/24. Prior static analysis techniques, which target specific misconfiguration patterns in particular protocols, will not detect the second error, as that requires a precise model of the semantics of OSPF, connected routes, static routes, and their interaction through redistribution. Batfish finds both errors proactively as violations of failure consistency and multipath consistency properties (discussed below), respectively. It can do this because it (a) statically analyzes configurations, and (b) derives a faithful model of the data plane from configurations.

3 An Overview of Batfish

We now overview our approach to static analysis of network configurations, as implemented in Batfish. Figure 3 shows the four stages of its workflow.

```
//Part 1a: Facts on OSPF interface costs
OspfCost(n1, int1_2, 1)
...(remaining OSPF interfaces)
//Part 1b: Facts on OSPF adjacencies
OspfNeighbors(n1, int1_2, n2, int2_1).
OspfNeighbors(n1, int1_3, n3, int3_1).
OspfNeighbors(n2, int2_3, n3, int3_2).
...(symmetric facts)

//Part 2: Rules that capture basic OSPF logic
BestOspfRoute(node, network, nextHop, nhIp, cost) <-
   OspfRoute(node, network, nextHop, nhIp, cost),
   MinOspfRouteCost[node, network] = cost.

MinOspfRouteCost[node, network] = minCost <-
   minCost = agg<<cost = min(cost)>>:
      OspfRoute(node, network, _, _, cost).

OspfRoute(node, network, nextHop, nextHopIp, cost) <-
   OspfNeighbors(node, nodeInt, nextHop, nextHopInt),
   InterfaceIp(nextHop, nextHopInt, nextHopIp),
   ConnectedRoute(nextHop, network, nextHopConnInt),
   OspfCost(node, nodeInt, nodeIntCost),
   OspfCost(nextHop, nextHopConnInt, nextHopIntCost),
   cost = nodeIntCost + nextHopIntCost.

OspfRoute(node, network, nextHop, nextHopIp, cost) <-
   OspfNeighbors(node, nodeIntCost, nextHop, nhInt),
   InterfaceIp(nextHop, nhInt, nextHopIp),
   OspfNeighbors(nextHop, _, hop2, _),
   BestOspfRoute(nextHop, network, hop2, _, subCost),
   node != secondHop,
   cost = subCost + nodeIntCost.
```

Figure 4: *A subset of the control plane model for the OSPF portion of the configuration in Figure 2.*

3.1 From Configuration to Data Plane

The first two stages of Batfish transform the given network configuration into a concrete data plane. Stage 1 generates a logical model of the control plane. This model compactly represents the network configuration and topology and the computation that the network routers carry out collectively to produce the data plane.

Our control plane model is defined in a variant of Datalog called LogiQL, which is the language of the LogicBlox database engine [10, 17]. Beyond basic Datalog, LogiQL supports integers, arithmetic operations, and aggregation (e.g., minimum).

A key challenge addressed in our work is faithfully encoding the semantics of a range of low-level configuration directives in a high-level, declarative language. As we detail below, the declarative nature of our control plane and the resulting data plane models provides a simple ontology of relations that operators can query to understand the provenance of errors. While imperative code could have provided this capability, our declarative implementation gives us this information for free.

As an example, Figure 4 shows a portion of the control plane model for the configuration in Figure 2. Part 1 of

[1] Such intermittent connectivity can go unnoticed. To prevent reordering, multipath routing typically maps packets with the same 5-tuple (source and destination addresses and ports, and the protocol identifier) to the same path. If a connection gets unlucky and is initially mapped to the dropping path, subsequent retries (with a different source port) will likely map it to the valid path, after which all packets will be delivered.

the model has logical facts that encode the configuration and topology information. In the figure, we show the OSPF-related information, namely the link costs and adjacencies. Part 2 has a generic set of rules that capture the semantics of the control plane for an arbitrary network. In the figure, we show some of the rules for OSPF routing. The first rule defines the best OSPF route to be the route with the minimum cost. The second rule defines the minimum cost by simply aggregating over all OSPF routes to find the minimal element. The last two rules effectively implement a shortest-path computation.

The second stage of Batfish takes an *environment* as an additional input, which facilitates performing "what if" analysis. The environment consists of the up/down status of each link in the network as well as a set of route announcements from each of the network's neighboring ASes. It is represented as a set of logical facts.

We derive the data plane by executing the LogiQL program that represents the control plane model and the environment. This execution is essentially a fixed point computation, i.e., all rules are fired iteratively to derive new facts, until no new facts are generated. The resulting data plane model includes the forwarding behavior of individual routers as logical facts that indicate whether a packet with certain headers should be dropped (e.g., `Drop(node, flow)`) or forwarded to a neighbor (e.g., `Forward(node, flow, neighbor)`). The data plane model also includes facts for all of the intermediate predicates used in the rules; this enables users to easily investigate the provenance of various aspects of the data plane. For instance, a particular `Forward` predicate may have been derived from a `BestOspfRoute` fact in the control plane model, meaning that the chosen route came from OSPF, and that fact in turn was derived from a particular set of OSPF link costs in the configuration.

Unlike prior static analysis techniques, the first two stages of Batfish analyze all aspects of network configuration that are relevant to the data plane, irrespective of the correctness properties of interest. The resulting data plane thus faithfully captures the forwarding behavior induced by the given configuration, topology, and environment (but see §3.3 for limitations).

3.2 From Data Plane to Configuration Errors

The last two stages of Batfish identify and localize configuration errors. In the third stage, we analyze one or more data planes to check desired correctness properties. The tool can check any property expressible as a first-order-logic formula over the relations that represent one or more data planes of interest. This is accomplished by translating the data-plane relations and the correctness property to the language of the Z3 constraint solver [20, 35], which then either verifies the property or provides one or more counterexamples, which consist of a concrete packet header and originating router.

In addition to user-specified properties, Batfish checks for traditional reachability properties such as the absence of black holes and loops, as well as three new properties that go beyond reachability to ensure correctness of paths through the network and their relation to one another (§4). Because the first two stages of Batfish are property-independent, we can generate the data planes of interest once and then check any number of properties over these data planes without having to re-create them.

The final stage helps operators understand property violations, in order to properly repair the network configuration. It works by logically simulating the behavior of counterexample packets through the network on top of our logical data plane model. As before, various logical facts will be produced during this simulation. Some of these facts directly provide provenance information to the user, such as the particular line of an ACL that caused the packet to be dropped. The user can also investigate additional provenance relationships by querying the full logical database, which contains facts about the control plane, the data plane, and their relationship, to understand why particular facts were generated.

To understand the process of uncovering the root cause of an error found by Batfish, consider the second error described for the example network in §2.2. Batfish detects this error as a *multipath inconsistency*. See §4 for the formal definition, but informally, it means that packets of a flow can be dropped along some paths but carried to destination along some others. This inconsistency is represented it by the following logical fact:

```
FlowMultipathInconsistent(Flow<src=n1, dstIp=10.0.0.0>)
```

The operator can then query the `FlowTrace` relation of Batfish, which produces a traceroute-like representation of the paths taken by the counterexample flow:

```
FlowTrace(Flow<src=n1, dstIp=10.0.0.0>,
   [n1:int1_2 -> n2:int2_1]:accepted])
FlowTrace(Flow<src=n1, dstIp=10.0.0.0>,
   [n1:int1_3 -> n3:int3_1]:nullRouted)
```

To understand why the flow was accepted by *n*2 but dropped by *n*3, the operator can then query the `FlowMatchRoute` relation to see which routes the flow matched at each router in the above paths:

```
FlowMatchRoute(Flow<src=n1, dstIp=10.0.0.0>, n1,
   Route<prefix=10.0.0.0/24, nextHop=n2, 10, ospfE2>)
```

```
FlowMatchRoute(Flow<src=n1, dstIp=10.0.0.0>, n1,
   Route<prefix=10.0.0.0/24, nextHop=n3, 10, ospfE2>)
FlowMatchRoute(Flow<src=n1, dstIp=10.0.0.0>, n2,
   Route<prefix=10.0.0.0/24, int=int2_5, connected>)
FlowMatchRoute(Flow<src=n1, dstIp=10.0.0.0>, n3,
   Route<prefix=10.0.0.0/24, DROP, static>)
```

Here we see that $n1$ has two external type-2 (redistributed, fixed-cost) OSPF routes to 10.0.0.0/24 with equal cost of 10. The first points to $n2$ where the network is directly-connected, and the second points to $n3$ which has a static discard route for the destination. To prevent the discard route at $n3$ from being active on $n1$, the operator may increase the exported cost of this route on $n3$ in line 4 of Figure 2.

3.3 Discussion

Since Batfish strives to model all aspects of configuration that impact forwarding, when checking for correctness our approach incurs no false positives and no false negatives; each identified error is a real violation of the checked property, and all violations are identified. However, this guarantee has three caveats from a pragmatic perspective. First, like other configuration analysis tools, we assume that routers behave as expected based on their configurations. We cannot catch errors due to bugs in router hardware or software (e.g., BGP implementation).

Second, Batfish analyzes a network under a given set of environments, which are a subset of all possible environments. Therefore, Batfish can miss errors that occur only in environments that the operator has not supplied. Further, operators may supply an infeasible environment to Batfish. For instance, the routing announcements from C1 and P1 in Figure 2 may be correlated in some complex way because those ASes are connected through a path that is not visible to our analysis. In this case, errors identified by Batfish may be spurious since a particular analyzed data plane might never occur in reality.

Finally, Batfish may encounter configuration features that are currently not implemented (e.g., the internal 'color' metrics of Juniper) but may influence local route selection. If that happens, the tool warns users that the guarantee may not hold. There is a qualitative difference, however, between the incompleteness of Batfish and of prior configuration analysis tools. Because Batfish uses the data plane as an intermediate representation, currently-unimplemented features can be mapped to this representation simply by adding logical rules to our control-plane model for how they impact forwarding. Because prior tools use custom intermediate representations or custom checkers, it may be difficult or impossible to use them to model and reason about some new features. Currently, Batfish models a rich enough subset of the configuration space (§6) to precisely analyze two large university networks.

4 Consistency Properties

Batfish can take as input any specification of intended network behavior and automatically check whether the network indeed behaves as expected. For instance, the operator might specify that the network should not carry packets from one particular neighboring AS to another. However, to simplify the task of finding potential errors, we also propose three safety properties that were motivated by discussions with network operators and require little or no input from users. These properties flag different forms of inconsistencies in the network behavior. Prior work on verification in several domains has shown that inconsistent behavior often points to bugs [6, 7].

Our properties are expressed using two auxiliary predicates which we define first. Let E be the environment used to generate the data plane model in Stage 2 of our pipeline. We define predicates $\mathsf{accepted}_E(H,S,D)$ and $\mathsf{dropped}_E(H,S,D)$, which hold if there is some path through the network for which header H is eventually accepted and dropped, respectively, at node D when injected into the network at node S. "Accepted" implies that the packet either reaches its destination or is forwarded outside the modeled network. We simulate packets as being sent along all equal-cost paths, so accepted and dropped are not mutually exclusive. It is straightforward to define these predicates in terms of the logical relations that comprise the data plane. Below we sometimes omit the last argument to the accepted predicate when it is irrelevant, as shorthand for the formula $\exists D$: $\mathsf{accepted}_E(H,S,D)$; a similar shorthand is used for the dropped predicate.

4.1 Multipath Consistency

Multipath consistency is a property that is relevant to networks that use multipath routing and it captures the following expected behavior: all packets with the same header should be treated identically in terms of being accepted or dropped, regardless of the path taken through the network. Formally, we say that the network with environment E exhibits *multipath consistency* if the following condition is true:

$$\forall H,S : \mathrm{accepted}_E(H,S) \Rightarrow \neg\mathrm{dropped}_E(H,S)$$

In other words, every packet is either accepted on all paths or dropped on all paths. A counterexample to this formula consists of a concrete packet header and source node such that it is possible for the header to be both accepted and dropped depending on the path taken.

4.2 Failure Consistency

Networks are typically designed to be tolerant to some number of faults. For example, a particular node or link may have been intended to be used as a backup for another node or link. However, it can be difficult for operators to reason about whether the network configuration is indeed as fault tolerant as intended.

We define a general notion for verifying fault tolerance of a network configuration. Let E' be the network environment identical to E but with a subset of links or nodes considered failed. This subset is drawn from the class of failures to which the network is designed to be fault tolerant (e.g., all single-link failures). We say that the network exhibits *failure consistency* between E and E' if the following condition is true:

$$\forall H,S : \text{accepted}_E(H,S) \Rightarrow \text{accepted}_{E'}(H,S)$$

A counterexample to this formula is a concrete packet header and source node such that the packet is accepted under environment E but dropped under E'. Of course, packets destined for any interface that is failed in E' should not be considered counterexamples to failure consistency. Thus, the full property definition, which we omit for simplicity, includes an extra condition that requires H to be destined for an active interface in E'.

4.3 Destination Consistency

Customer ASes of a given network are often expected to have disjoint IP address spaces, sometimes assigned by the network itself. In such cases, the intended network configuration is to allow a customer AS to only send route announcements for its own address space, ensuring that it only receives packets destined to itself. Our destination consistency property captures this expectation. Let E be the network environment with only customer ASes (i.e., provider and peer AS nodes are considered failed) and E' be an identical environment but with all links to a customer AS C considered failed. Then we say that the network exhibits *destination consistency* for C if the following condition is true:

$$\forall H,S : \forall D \in C :$$
$$\text{accepted}_E(H,S,D) \Rightarrow \neg\text{accepted}_{E'}(H,S)$$

In other words, any packet that is accepted by some node D in the AS C should not be accepted once C is removed.

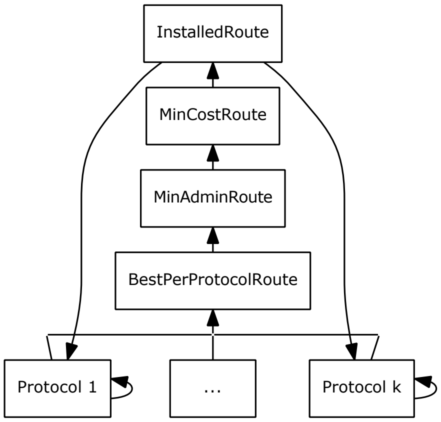


Figure 5: *Information flow for computation of the RIB.*

A counterexample to this formula consists of a concrete packet header, source node, and destination node D in AS C such that the packet is accepted at D under environment E and is accepted somewhere in E'.

5 The Four Stages of Batfish

In this section we present details on each of the four stages in the Batfish pipeline (Figure 3).

5.1 Modeling the Control Plane

Batfish's first stage takes configuration files and network topology as input, and it outputs a control plane model that captures the distributed computation performed by the network. The input information is first parsed into an intermediate data structure, which is then translated into a set of logical *facts*, each associated with a particular *relation*. For example, `SetIpInt(Foo, f0/1, 1.2.3.4, 24)` says interface `f0/1` of node `Foo` has IP address 1.2.3.4 with a 24-bit subnet mask.

These base facts are combined with a set of logical rules that specify how to infer new facts. These rules capture route computation for various protocols. In more detail, each node may be configured to run one or more routing protocols (e.g., OSPF, BGP, etc.). At each node, each protocol iteratively computes its best route to each destination in the network using information learned from neighbors. The available routes to destinations are stored in a routing information base (RIB). While RIB formats vary, a typical RIB entry minimally contains a destination network, the IP address of the next hop for that network, and the protocol that produced the entry. When multipath routing is being used, multiple best routes may be selected for a destination.

Our routing rules capture the process by which RIB entries are generated at each node. Figure 5 shows how we model this process. The model consists of four main relations, each representing a set of routes, and the edges denote the dependencies among these sets.

`BestPerProtocolRoute` is the set of routes that are optimal according to the rules of one of the routing protocols. Protocol-specific rules are defined in terms of a set of relations that represent facts from the configuration and topology information. For example, the OSPF rules shown earlier depend on configured link costs. As Figure 5 shows, our model is modular with respect to such protocols, and adding a new protocol simply requires rules for producing its optimal routes.

`MinAdminRoute` is the subset of `BestPerProtocolRoute` with only routes that have minimal *administrative distance*, a protocol-level configuration parameter. That is, `MinAdminRoute` contains a route *R* to destination *D* from `BestPerProtocolRoute` if the protocol that produced *R* has an administrative distance no higher than that of any other protocol that produced a route to *D*.

`MinCostRoute` is the subset of `MinAdminRoute` with only those routes that have minimal *protocol-specific cost*. That is, `MinCostRoute` contains a route *R* to destination *D* from `MinAdminRoute` if *R* has a protocol-specific cost no higher than that of any other route to *D* in `MinAdminRoute`.

`InstalledRoute` is the set of routes that are selected as best for the node. This set is identical to `MinCostRoute` but is given a new name for clarity.

In general, the set of candidate routes produced by a routing protocol may depend on the current state of the RIB, as well as the internal state of that protocol and the latest messages it has received. We have an edge from `InstalledRoute` to each protocol to illustrate the dependence on previous state, and also to model any redistribution of installed routes from one protocol to another. Thus, these edges signify that producing the RIB requires computing the fixed point of the function that generates the next intermediate state of the RIB.

Figure 6 shows key LogiQL rules for the relations in Figure 5. The `agg` keyword refers to an aggregation; in this case we are finding the tuples of a relation whose aggregated variable is minimal among all the tuples. In addition to such generic rules, we implement LogiQL rules for several routing protocols, and as noted above, a new protocol can be added completely modularly.

```
InstalledRoute(node, network, nextHop,
 nextHopIp, admin, cost, protocol) <-
  MinCostRoute(node, network, nextHop,
   nextHopIp, admin, cost, protocol)

MinCostRoute(node, network, nextHop,
 nextHopIp, admin, minCost, protocol) <-
  minCost = MinCost[node, network],
  MinAdminRoute(node, network, nextHop,
   nextHopIp, admin, minCost, protocol)

MinCost[node, network] =  minCost <-
   agg<<minCost = min(cost)>>
   MinAdminRoute(node, network, _, _, _, cost, _)

MinAdminRoute(node, network, nextHop,
 nextHopIp, minAdmin, cost, protocol) <-
  minAdmin = MinAdmin[node, network],
  BestPerProtocolRoute(node, network,
   nextHop, nextHopIp, minAdmin, cost,
   protocol)

MinAdmin[node, network] =  minAdmin <-
   agg<<minAdmin = min(admin)>>
   BestPerProtocolRoute(node, network,
    _, _, admin, _, _).
```

Figure 6: *LogiQL code for route-selection.*

5.2 Building the Data Plane

The data plane of the network is the forwarding information base (FIB) for each node. The FIB determines an appropriate action to take when a packet reaches a particular interface. For the purposes of this paper, that action is either to forward the packet out of one or more interfaces, to accept the packet, or to drop the packet. The second stage of Batfish generates one data plane per user-specified environment.

In Batfish, the FIB for a node consists of the node's RIB, the configured ACLs for the node's interfaces, and rules for using these items to forward traffic. The data-plane generator starts by simply executing the LogiQL program that is the output of Stage 1, which is the control-plane model, to produce the RIB for each node. Before doing so, LogiQL facts to represent the provided environment are added to the model. Specifically, the facts indicate which interfaces in the network are up, allowing us to model network failures, and which routes are being advertised by neighboring networks.

A LogiQL program consisting of a set of base facts and rules is executed as follows. When a rule body (to the right of <− in Figure 6) is satisfiable by existing facts, a new fact is derived and added to the relation in the rule head (to the left of <−). This process repeats until quiescence. At this point, the facts in the `InstalledRoute` relation represent the RIB for each node. We then represent the FIB as a new set of logical rules that make forwarding decisions, given the RIB information as well as the per-interface ACLs, which were converted to logical facts in Stage 1.

The rules for the FIB are as follows. When a packet arrives on an interface, the rules first check whether the interface has an incoming ACL. If so, and if the packet's header is not allowed by that ACL, the packet is dropped. Otherwise, if the destination IP address of the header is assigned to any interface of the node, then the packet is accepted. Otherwise, the rules check the RIB for entries with networks that are longest-prefix matches for the destination IP address of the header. For each such route, the interface corresponding to that route's next hop is determined as follows: if the route is directly connected on an interface, that interface is selected. Otherwise, the rules use the next hop of the route that is a longest prefix match for the address of the original next hop, recursively, until a directly connected route is found. Finally, the packet is forwarded out that interface if the interface's outgoing ACL permits it, and dropped otherwise.

5.3 Property Checking

After Stage 2, users have access to the full power of LogiQL to ask queries about both the control and data planes. Moreover, these queries can directly employ the relations in our high-level conceptual model. For example, users can query the `BestOspfRoute` relation to find the best OSPF route(s) to a particular destination on a particular node. Further, by employing multiple relations in a query users can easily obtain even richer information, such as the set of all BGP advertisements for a particular prefix that were rejected by an incoming route-map on at least two nodes. In this way, users can interactively investigate various aspects of the network's forwarding behavior as well as their provenance.

In addition to user-directed exploration, Batfish supports systematic checking of correctness properties, to find errors and to prove their absence. By default it checks the properties in §4, but operators can supply additional properties, expressed as first-order formulas over the relations in our data plane model. Depending on the property, Batfish requires one or more data plane models that differ in their environment (e.g., link failures).

Batfish uses Network Optimized Datalog (NoD) [20], a recent extension of the Z3 constraint solver, to identify violations of correctness properties. The properties we check are decidable and can be expressed precisely in NoD and Z3, so Batfish is guaranteed to find a counterexample if one exists, modulo resource limitations. In the rest of the paper, we use *NoD* to refer to the NoD extension to Z3 and use *Z3* to refer to the vanilla Z3 solver (which we also use). To check a property P, we ask NoD if its negation $\neg P$ is satisfiable in the context of the given data plane models. If not, the property holds. If so, NoD provides the complete boolean formula expressing how to satisfy the negation of the property. This formula is a set of constraints on a packet header and the interface at which the packet is injected into the network. We then query Z3 to solve these constraints, thereby producing a concrete counterexample that violates P.

5.4 Provenance Tracking

The final stage of Batfish helps users to localize the root cause of identified property violations. First, each counterexample from the previous stage is converted into a concrete test flow in terms of our LogiQL representation of the data plane. Then, this test flow is "injected" into our logical model, causing LogicBlox to populate relevant relations with facts that indicate the path and behavior of the flow through the network. Many of the produced facts include explicit provenance information, and as demonstrated in §3.2, users can iteratively query the populated relations to map errors back to their sources in the configuration files.

6 Implementation

We implemented Batfish using Java and the Antlr [27] parser generator. Its source comprises 21,504 lines of Java code, 13,214 lines of Antlr code across 2,410 grammar rules, and 5,696 lines of LogiQL code across 386 relations. The bulk of the Java and Antlr code corresponds to Stage 1 of Batfish, which converts configurations to LogiQL facts.

To manage the complexity supporting diverse configuration languages and diverse directives within a language (with overlapping functionality), we devised a vendor- and language-agnostic representation for control plane information. We first translate the original configuration files to our representation, and the rest of the analysis uses this representation exclusively. Therefore, support for new languages or directives can be added by implementing appropriate translation routines, without having to change the core analysis functionality.[2,3]

We currently support configuration languages of Cisco

[2] This analysis structure is akin to LLVM [16], which facilitates analysis of code written in multiple programming languages by first converting the code to a common representation.

[3] We hope that in the future router vendors would supply the translation routines as they best understand the semantics of their languages and directives.

IOS, Cisco NXOS, Juniper JunOS, Arista, and Quanta. Our models of the control and data plane are rich enough to capture the behavior of many real, large networks. We faithfully model static routes, connected networks, interior gateway protocols (e.g., OSPF, including areas), BGP, redistribution of routes between routing protocols, firewall rules, ACLs, multipath routing, VLANs, forwarding based on longest-prefix matching, and policy routing. We currently do not model MPLS [30] or packet modification (e.g., NATs).

A semantic mismatch in encoding configuration directives in LogiQL is for regular expression matching. Such matching may be used for BGP communities and AS-paths but is not supported by LogiQL. We implement community-matching by precomputing the result of the match for all communities mentioned in configuration files and the environment. This strategy does not work for AS-path matching because AS-paths are lists (where order matters; communities are sets) and all possible AS-paths are not known statically.

Based on the observation that regular expressions in configuration files tend to be simple, we implement matching only for regular expressions that match sub-paths of size two or less. For example, if the regular expression is `.*[5-10][10-15].*`, we use LogiQL predicates that are true when the AS-path, encoded as a LogiQL list, has an item between 5 and 10 followed by one between 10 and 15. This limited support sufficed for the networks we analyzed, but it can be extended to longer subpaths.

7 Evaluation

"P.S. WRT the prefix that was dual assigned from yesterday, one of my NOC [network operations center] guys stopped by today to ask what voodoo I was using to find such things :)"

– email from the head of the Net1 NOC

To evaluate Batfish, we used it on the configuration of two large university networks with disparate designs. We call them Net1 and Net2 in this paper as the operators requested anonymity. We aim to ascertain whether Batfish can scale to handle such real-world networks and whether it can find configuration errors in them.

7.1 Analyzed Networks

We analyzed recent network configurations from Net1 and Net2. They were working, stable configurations for which the operators were unaware of any bugs.

Net1 The routing design of Net1 uses BGP internally, modeling academic departments and a few other organizational entities (e.g., libraries, dorms) as ASes. The campus core network consists of 21 routers in 3 tiers: 3 border routers, 5 core routers, and 13 distribution routers. All routers run OSPF for internal connectivity. The border routers have eBGP peering sessions with two provider ASes and iBGP peering sessions with the core routers. The distribution routers have eBGP peering sessions with 52 internal ASes which are treated as customers of the core network. By design, each department AS is expected to have redundant peering connections with the Net1 core network, and each department should have its own distinct address space. Distribution routers also have iBGP peering sessions with the core routers.

As mentioned earlier, the environment for analysis of a network includes the route announcements from neighboring ASes. We used a single set of route announcements for all of the experiments on Net1. These route announcements were defined by creating stub configuration files for a new set of routers that represent Net1's BGP peers; this has the effect of populating the appropriate relations of our control plane model in Stage 1. The provider AS routers were simply configured to advertise a default route (i.e., the AS is willing to carry any traffic). The department AS routers were configured to advertise every network that their Net1 peer would accept but drop all traffic that was not destined to their own delegated address space. This approach ensures that we do not assume department ASes are "well behaved" when checking for vulnerabilities in Net1. Including these new routers, the topology we analyzed has 75 nodes.

Net2 The routing design of Net2 is qualitatively different. It employs VLANs to model the network as a large layer-2 domain. The network consists of 17 routers, of which three are core routers on the main campus and the rest interconnect the main campus with satellite campuses. All routers run OSPF for internal connectivity.

Since Net2 does not use BGP internally, we did not model the network's neighbors explicitly, as was done for Net1. Rather, the environment we used contained no route announcements from neighbors, and the analyzed topology included just the original 17 nodes.

7.2 Experiments

We checked for each consistency property in §4.

Multipath Consistency This property was encoded as a logical formula described in §4. We posed one NoD query pair per source node in the network, which asks for the existence of a header exhibiting a multipath inconsistency when injected at the given node. Whenever such a header was identified, it was fed into Stage 4 of Batfish, which produced provenance information that pointed to the source of the inconsistency in the original configurations. We then patched the configurations and iterated until all queries were unsatisfiable.

Failure Consistency For this experiment, we generated the data plane corresponding to no failures as well as one data plane for each possible failure of a single (non-generated) interface (199 for Net1, 279 for Net2). We used NoD to separately obtain constraints on packets that are accepted in the no-failure scenario and constraints on packets that are not accepted in each failure scenario, again with separate queries per each possible source node. Finally we asked Z3 to find a concrete header satisfying the constraints of both the no-failure scenario and the failure scenarios, for each possible failure scenario and each source node in the network.

Destination Consistency For Net1 we generated 53 separate data planes: one corresponding to the unchanged configurations and one corresponding to the removal of each of Net1's 52 customer ASes. We excluded the provider ASes from this analysis altogether, since in general a provider may appear to provide an alternate path to any prefix that is part of a separate AS. We then used NoD and Z3 in the same way as described above for failure consistency, to identify headers that are accepted in the original data plane and also accepted after the destination's associated peer is removed from the network.

Destination consistency is not applicable to Net2, since it has no customer ASes.

7.3 Results

Batfish found a variety of bugs in both networks. Many of the concrete counterexamples it reported had different headers but were due to the same underlying configuration issue or an analogous issue on a different router. This makes counting the number of distinct issues somewhat difficult, so we provide two different metrics. First, we count one *bug* for each inconsistency related to an explicitly declared space of packet headers or source IPs in the network configuration. Second, we group bugs of a similar nature into *bug classes*. For instance, if a prefix list is incorrectly defined in two routers, we may find two unique bugs but we consider them to be in the same class. In general, the relationship between bugs and bug classes is complex: a change to a network configuration may remove one, two, or more bugs from the same class.

Table 1 summarizes our results for both the number of bugs and bug classes (in parenthesis). We reported each property violation with its provenance information to the operators. The "Total violations" column gives the number of bugs and bug classes we reported. The "Undesired behaviors" column contains the subset of total violations that the operators confirmed caused undesired behaviors in the network. The only difference in these columns occurs for failure consistency. As explained below, this difference is not due to false positives in the analysis but reflects an intentional lack of fault tolerance in portions of the network or lack of fidelity in modeling network topology. "Fixed violations" is the subset of undesired violations that were fixed after we reported them. Not all behaviors that were confirmed as undesired by operators could be immediately fixed because the change was complex or the operators feared collateral damage.

		Total violations	Undesired behaviors	Fixed violations
Net1	Multipath	32 (4)	32 (4)	21 (3)
	Failure	16 (7)	3 (2)	0 (0)
	Destination	55 (6)	55 (6)	1 (1)
Net2	Multipath	11 (3)	11 (3)	11 (3)
	Failure	77(26)	18(7)	0(0)

Table 1: *Number of bugs (bug classes) for each property.*

Finally, a fix to a configuration may eliminate violations of multiple consistency properties. For instance, we see cases in which a fix that the operator applied for multipath consistency also removed some violations of failure consistency. In Table 1, we count such violations only once (for the property listed highest).

7.3.1 Understanding the discovered bugs

We now provide insight into issues that were uncovered by Batfish.

Multipath Consistency For Net1, a serious issue we found was a typo in the name of a prefix list on a Cisco router used to filter advertisements from one of the departments. The semantics for an undefined prefix list are to accept all advertisements. We found that this bug allowed the department to partially divert all traffic destined to other Net1 departments, as well as all traffic destined to arbitrary Internet addresses from any department. This bug was confirmed and fixed by the operators.

We show a sample of the provenance information for this bug below for a single hijacked prefix:

```
FlowAccepted(Flow<srcNode=nS, dstIp=10.0.0.1>, nV)
FlowDeniedIn(Flow<srcNode=nS, dstIp=10.0.0.1>, nA,
   Ethernet0, filter, 4)
FlowMatchRoute(Flow<srcNode=nS, dstIp=10.0.0.1>, nS,
   Route<prefix=10.0.0.0/24, nextHop=nA, ibgp>)
FlowMatchRoute(Flow<srcNode=nS, dstIp=10.0.0.1>, nS,
   Route<prefix=10.0.0.0/24, nextHop=nV, ibgp>)
```

Here, *nA* is an adversarial department that can send arbitrary advertisements, and *nV* is a victim department whose network has been hijacked. This indicates that some source node *nS* has iBGP routes to the victim's prefix through either the victim or the adversary. This would not be possible if advertisements from the adversary were filtered properly.

We also discovered three bug classes in which ACLs intended for the same department on different routers did not match. Two of these bug classes were fixed. The third bug class was confirmed as a problem, but the operators did not immediately fix it. The network operator stated that the ACLs in those cases matched the prefixes the peer wanted to announce at each connection point. He further commented, however, that should the peer change where these prefixes are announced without notice, traffic could get dropped. Therefore, he decided to change the policy in the future to accept all peer-delegated prefixes at each connection point, leaving it to the peer to decide what gets announced where.

For Net2, all of the multipath consistency bugs were due to inconsistent handling of routes redistributed into OSPF. In some cases, a connected route and a null static route (one configured to drop traffic for a prefix) for the same prefix would be redistributed by two different routers, and both of these routers would be installed as next hops for this prefix by a third adjacent router (§2.2). In the other cases, two routers would redistribute connected routes to a link they shared, but the path through one router would allow some traffic while the path through the other might deny it due to the ACLs applied on that path.

Failure Consistency For Net1, all of the failure consistency violations that Batfish found occurred when the interface that connected a department peer was failed. This situation indicates that the peer's only connection to the core network was through the interface disabled for the experimental run, which implies an absence of fault tolerance. The network operator reported that several such cases were known and due to economic reasons. The peer could not afford to maintain multiple links, or laying another line would be prohibitively expensive. We did not count these cases as "Undesired behaviors."

For Net2 Batfish found 26 bug classes for failure consistency, as shown in Table 1. But 19 were not deemed as undesired behaviors by the network operator. 5 were due to a bad assumption in how we currently model VLANs. We assume one-to-one mapping between logical VLAN interfaces and physical interfaces, but in reality the relationship was one-to-many for some VLANs, which led Batfish to underestimate fault tolerance. 14 bug classes represented intentional absence of fault tolerance. In 6 of them, providing backup paths was deemed prohibitively expensive. Interestingly, in 8 cases, backup paths existed but certain types of traffic were not allowed to use it.

Batfish found 7 bug classes that represented unexpected lack of fault tolerance. In 5 cases, it was due to VLAN implementation using a single physical interface. In the rest, only a single link served certain paths, which surprised the operators. These inconsistencies could not be fixed immediately because the solution needed new hardware and links in addition to configuration changes.

Destination Consistency For Net1, we found one bug (class) which the operators fixed: advertisements for a particular prefix were erroneously permitted from both the dorms and an academic department. This situation allowed the dorms to hijack the department's traffic.

The other discovered cases of destination consistency were confirmed by the operator as undesirable but were also known. These were cases in which advertisements for a prefix were permitted from several peers, but these peers actually fell under one administrative unit; they were separated into multiple ASes because of legacy considerations, and/or an unwillingness on the part of the peer operators to disturb a working system. The operator noted that ideally they would all fall under a single AS and wants to start consolidating them. Thus, the discovered violations represent fragility in the face of changes on the other end, but should not disrupt traffic as is.

7.4 Performance benchmarks

The time to analyze a network using Batfish depends on the size and complexity of the network and the correctness properties checked, as well as the performance of third-party tools such as NoD and LogicBlox. But we provide some insight by reporting on what we observed for our networks. We focus on the second and third stages of Batfish, as the other two stages take relatively little time (under a minute).

First consider multipath consistency. On an Intel E5-2698B VM, data plane generation (Stage 2) takes 238 (37) minutes for Net1 (Net2). Checking multipath consistency (Stage 3) requires making 75 (17) NoD and Z3 query pairs, each component of which takes under 90 seconds on a single core. Each query is completely independent of the others, so Batfish performs them in parallel. A significant portion of the time to compute the data plane for Net1 is due to the large number of routes advertised by the generated department configurations; we believe this computation can be optimized significantly.

Failure consistency is the most onerous of our properties to check, since it requires one data plane per failure case of interest. There are 199 (279) such failure cases for Net1 (Net2); each can be checked independently. With an optimal number of processing nodes, i.e. 1 per data-plane, the computation time will not be appreciably more than that for multipath consistency.

Operators that have access to only modest hardware resources can use Batfish as follows. Before applying a configuration change, they can check for only multipath consistency and other properties that do not require additional data planes. This provides important correctness guarantees for the common case of no failures. Then, after applying the configuration change, the operators can continue to check for other properties in the background.

8 Related Work

Our work builds on several threads of prior work. One such thread is the static analysis of network configurations, which, as detailed in §1, has focused on specific aspects of the configuration or specific properties, enabling customized solutions [2, 7, 11, 24, 25, 34]. For instance, rcc [7] and IP Assure [24] perform a range of checks that pertain to particular protocols or configuration aspects (e.g., the two ends of an OSPF link are in the same area, link MTUs match, the two ends of an encrypted tunnel use the same type of encryption-decryption). While violations identified by such static analysis tools likely represent poor practices, the tools cannot, unlike Batfish, indicate whether or how violations impact the network's forwarding. On the other hand, for a violation that occurs only in specific environments (e.g., when certain kinds of external routes are injected in the network), Batfish can detect it only when given a concrete instance of one of these environments, but a specialized tool for checking particular properties may be able to uncover such a violation even without these concrete inputs by leveraging specific characteristics of those properties.

Closer to our work are approaches that directly model network behavior from its configuration. For example, Feamster et al. [8] develop a tool to compute the outcome of BGP route selection for an AS. Xie et al. [33] outline how to infer reachability sets, which are sets of packets that can be properly carried between a given source and destination node in the network. Benson et al. [4] extend this notion of reachability to assess the complexity of a network. Batfish is similar in spirit but broader in scope, handling all aspects of configuration that affect forwarding and producing a complete data plane.

The C-BGP [28] and Cariden [5] tools also generate a data plane from network configuration, but they use an imperative, simulation-based approach, and focus on specific configuration aspects (BGP and traffic engineering, respectively). We employ a declarative approach, which provides a way to tractably reason about all aspects of the configuration. More importantly, Batfish provides provenance information and the ability to query intermediate control plane relations.

Anteater [22] and Hassel [14] analyze data plane snapshots, obtained by pulling router FIBs and parsing portions of configuration that map directly to forwarding state (e.g., ACLs). More recent data plane analysis tools focus on SDNs and faster computations [13, 15, 20, 37]. By starting from the network configuration, Batfish can find forwarding problems proactively and enable "what if" analysis across different environments. However, data-plane snapshot analysis is not rendered expendable by our approach. Such analysis can find forwarding problems due to router software bugs, while we assume that the router faithfully implements the configurations. Thus, both types of analyses are valuable in the network verification toolkit.

Batfish employs NoD [20] to perform data-plane analysis in Stage 3 of its pipeline. We picked NoD because it had better performance and usability than prior tools. NoD has been used by its creators for "differential reachability" queries, one of which is analogous to our notion of multipath consistency. Their queries and our properties were developed independently.

9 Conclusions

We develop a new approach to analyze network configuration files that can flag a broad range of forwarding problems proactively, without requiring the configuration to be applied to the network. For two large university networks, our instantiation of the approach in the Batfish tool found many misconfigurations that were quickly fixed by the operators. Our approach is fully declarative and derives, from low-level network configurations, logical models of the network's control and data planes. We believe that these models are useful beyond finding configuration errors, for instance, to migrate a network toward high-level programming frameworks while faithfully preserving its existing policies.

Acknowledgments We thank the NSDI reviewers and our shepherd Nick Feamster for feedback on this paper, Nuno Lopes and Nikolaj Bjorner for help with NoD and Z3, Martin Bravenboer for help with LogicBlox, and the operators of the networks we analyzed for their assistance and feedback. This work is supported in part by the National Science Foundation award CNS-1161595.

References

[1] Batfish. `http://www.batfish.org`, February 26, 2015.

[2] E. Al-Shaer and H. Hamed. Discovery of Policy Anomalies in Distributed Firewalls. In *Proceedings of the Twenty-third Annual Joint Conference of the IEEE Computer and Communications Societies*, INFOCOM '04, New York, NY, USA, March 2004. IEEE.

[3] M. Anderson. Time Warner Cable Says Outages Largely Resolved. New York, NY, USA, August 2014. Associated Press.

[4] T. Benson, A. Akella, and D. Maltz. Unraveling the Complexity of Network Management. In *Proceedings of the 6th USENIX Symposium on Networked Systems Design and Implementation*, NSDI '09, Berkeley, CA, USA, April 2009. USENIX Association.

[5] Cariden Technologies, Inc. IGP Traffic Engineering Case Study, October 2002.

[6] D. R. Engler, D. Y. Chen, and A. Chou. Bugs as Inconsistent Behavior: A General Approach to Inferring Errors in Systems Code. In *Proceedings of the 18th ACM Symposium on Operating Systems Principles*, SOSP '01, New York, NY, USA, October 2001. ACM.

[7] N. Feamster and H. Balakrishnan. Detecting BGP Configuration Faults with Static Analysis. In *Proceedings of the 2nd USENIX Symposium on Networked Systems Design and Implementation*, NSDI '05, Berkeley, CA, USA, May 2005. USENIX Association. Tool source code at `https://github.com/noise-lab/rcc/`.

[8] N. Feamster, J. Winick, and J. Rexford. A Model of BGP Routing for Network Engineering. In E. G. C. Jr., Z. Liu, and A. Merchant, editors, *Proceedings of the Joint International Conference on Measurement and Modeling of Computer Systems*, SIGMETRICS '04, New York, NY, USA, June 2004. ACM.

[9] N. Foster, R. Harrison, M. J. Freedman, C. Monsanto, J. Rexford, A. Story, and D. Walker. Frenetic: A Network Programming Language. In *Proceedings of the 16th ACM SIGPLAN International Conference on Functional Programming*, ICFP '11, New York, NY, USA, September 2011. ACM.

[10] S. S. Huang, T. J. Green, and B. T. Loo. Datalog and Emerging Applications: An Interactive Tutorial. In *Proceedings of the 2011 ACM SIGMOD International Conference on Management of Data*, SIGMOD '11, New York, NY, USA, June 2011. ACM.

[11] K. Jayaraman, N. Bjorner, G. Outhred, and C. Kaufman. Automated Analysis and Debugging of Network Connectivity Policies. Technical Report MSR-TR-2014-102, Microsoft Research, July 2014.

[12] C. R. Kalmanek, S. Misra, and Y. R. Yang, editors. *Guide to Reliable Internet Services and Applications*. Springer, 2010.

[13] P. Kazemian, M. Chang, H. Zeng, G. Varghese, N. McKeown, and S. Whyte. Real Time Network Policy Checking Using Header Space Analysis. In *Proceedings of the 10th USENIX Symposium on Networked Systems Design and Implementation*, NSDI '13, Berkeley, CA, USA, April 2013. USENIX Association.

[14] P. Kazemian, G. Varghese, and N. McKeown. Header Space Analysis: Static Checking for Networks. In *Proceedings of the 9th USENIX Symposium on Networked Systems Design and Implementation*, NSDI '12, Berkeley, CA, USA, April 2012. USENIX Association.

[15] A. Khurshid, X. Zou, W. Zhou, M. Caesar, and P. B. Godfrey. VeriFlow: Verifying Network-wide Invariants in Real Time. In *Proceedings of the 10th USENIX Symposium on Networked Systems Design and Implementation*, NSDI '13, Berkeley, CA, USA, April 2013. USENIX Association.

[16] The LLVM compiler infrastructure. `http://llvm.org/`, February 26, 2015.

[17] LogicBlox, Inc. LogicBlox 4 Reference Manual. `https://developer.logicblox.com/content/docs4/core-reference/html/index.html`, February 26, 2015.

[18] B. T. Loo, T. Condie, M. Garofalakis, D. E. Gay, J. M. Hellerstein, P. Maniatis, R. Ramakrishnan, T. Roscoe, and I. Stoica. Declarative Networking: Language, Execution and Optimization. In *Proceedings of the 2006 ACM SIGMOD International Conference on Management of Data*, SIGMOD '06, New York, NY, USA, June 2006. ACM.

[19] B. T. Loo, J. M. Hellerstein, I. Stoica, and R. Ramakrishnan. Declarative Routing: Extensible Routing with Declarative Queries. In *Proceedings of the ACM SIGCOMM 2005 Conference*, SIGCOMM '05, New York, NY, USA, June 2005. ACM.

[20] N. P. Lopes, N. Bjorner, P. Godefroid, K. Jayaraman, and G. Varghese. Checking Beliefs in Dynamic Networks. In *Proceedings of the 12th USENIX Symposium on Networked Systems Design and Implementation*, NSDI '15, Berkeley, CA, USA, May 2015. USENIX Association.

[21] R. Mahajan, D. Wetherall, and T. Anderson. Understanding BGP Misconfiguration. In *Proceedings of the ACM SIGCOMM 2002 Conference*, SIGCOMM '02, New York, NY, USA, August 2002. ACM.

[22] H. Mai, A. Khurshid, R. Agarwal, M. Caesar, P. B. Godfrey, and S. T. King. Debugging the Data Plane with Anteater. In *Proceedings of the ACM SIGCOMM 2011 Conference*, SIGCOMM '11, New York, NY, USA, August 2011. ACM.

[23] J. Moy. OSPF Version 2. RFC 2328, RFC Editor, April 1998.

[24] S. Narain, R. Talpade, and G. Levin. *Guide to Reliable Internet Services and Applications*, chapter Network Configuration Validation. In Kalmanek et al. [12], 2010.

[25] T. Nelson, C. Barratt, D. J. Dougherty, K. Fisler, and S. Krishnamurthi. The Margrave Tool for Firewall Analysis. In *Proceedings of the 24th Large Installation System Administration Conference*, LISA '10, Berkeley, CA, USA, November 2010. USENIX Association.

[26] T. Nelson, A. D. Ferguson, M. J. Scheer, and S. Krishnamurthi. Tierless Programming and Reasoning for Software-Defined Networks. In *Proceedings of the 11th USENIX Symposium on Networked Systems Design and Implementation*, NSDI '14, Berkeley, CA, USA, April 2014. USENIX Association.

[27] T. J. Parr and R. W. Quong. ANTLR: A Predicated-LL (k) Parser Generator. *Software: Practice and Experience*, 25(7), 1995.

[28] B. Quotin and S. Uhlig. Modeling the Routing of an Autonomous System with C-BGP. *IEEE Network: The Magazine of Global Internetworking*, 19(6), 2005.

[29] Y. Rekhter, T. Li, and S. Hares. A Border Gateway Protocol 4 (BGP-4). RFC 4271, RFC Editor, January 2006.

[30] E. Rosen, A. Viswanathan, and R. Callon. Multiprotocol Label Switching Architecture. RFC 3031, RFC Editor, January 2001.

[31] S. Shenker. The Future of Networking, and the Past of Protocols. Open Networking Summit, April 2012.

[32] O. Tange. GNU Parallel - The Command-Line Power Tool. *;login: The USENIX Magazine*, 36(1), February 2011.

[33] G. G. Xie, J. Zhan, D. A. Maltz, H. Zhang, A. Greenberg, G. Hjalmtysson, and J. Rexford. On Static Reachability Analysis of IP Networks. In *Proceedings of the Twenty-fourth Annual Joint Conference of the IEEE Communications Society*, volume 3 of *INFOCOM '05*, New York, NY, USA, March 2005. IEEE.

[34] L. Yuan, H. Chen, J. Mai, C.-N. Chuah, Z. Su, and P. Mohapatra. FIREMAN: A Toolkit for Firewall Modeling and Analysis. In *Proceedings of the 2006 IEEE Symposium on Security and Privacy*, New York, NY, USA, May 2006. IEEE.

[35] Z3 theorem prover. `https://z3.codeplex.com/` (opt branch), February 26, 2015.

[36] H. Zeng, P. Kazemian, G. Varghese, and N. McKeown. A Survey on Network Troubleshooting. Stanford HPNG Technical Report TR12-HPNG-061012, Stanford University, June 2012.

[37] H. Zeng, S. Zhang, F. Ye, V. Jeyakumar, M. Ju, J. Liu, N. McKeown, and A. Vahdat. Libra: Divide and Conquer to Verify Forwarding Tables in Huge Networks. In *Proceedings of the 11th USENIX Symposium on Networked Systems Design and Implementation*, NSDI '14, Berkeley, CA, USA, April 2014. USENIX Association.

Analyzing Protocol Implementations for Interoperability

Luis Pedrosa[†] *Ari Fogel*[‡] *Nupur Kothari*[*]
Ramesh Govindan[†] *Ratul Mahajan*[*] *Todd Millstein*[‡]

University of Southern California[†] University of California, Los Angeles[‡] Microsoft[*]

Abstract— We propose PIC, a tool that helps developers search for non-interoperabilities in protocol implementations. We formulate this problem using intersection of the sets of messages that one protocol participant can send but another will reject as non-compliant. PIC leverages symbolic execution to characterize these sets and uses two novel techniques to scale to real-world implementations. First, it uses *joint symbolic execution*, in which receiver-side program analysis is constrained based on sender-side constraints, dramatically reducing the number of execution paths to consider. Second, it incorporates a search strategy that steers symbolic execution toward likely non-interoperabilities. We show that PIC is able to find multiple previously unknown non-interoperabilities in large and mature implementations of the SIP and SPDY (v2 through v3.1) protocols, some of which have since been fixed by the respective developers.

1 Introduction

Nodes in distributed systems communicate using protocols such as TCP, HTTP, SPDY, and SIP. For robust operation, it is critical that the implementations of these protocols interoperate effectively, i.e., they should be able to correctly parse and interpret messages sent by each other.

Protocol interoperability is difficult to engineer and ensure because the protocols are complex, and often contain many mandatory and optional features. Moreover, protocol standards documents can be imprecise or ambiguous, leading different developers to make different implementation choices about protocol interactions and semantics of message header formats and values. Thus, in practice, it is neither possible nor sufficient to simply "certify" that an implementation adheres to a standard. Instead, pairs of implementations must be pitted against one another to test for interoperability.

Today, protocol developers spend significant manual effort [17, 18, 30, 31, 36] in testing interoperability. They identify test inputs that can test specific features of the protocol (e.g., options negotiation). But, because the space of test inputs is large, such manual testing is often incomplete. As a result, interoperability issues continue to frustrate developers, as well as users, even years after protocols have been fully deployed [35, 39].

Our goal is to automate the search for test inputs that trigger non-interoperabilities between two implementations. In this paper, we formulate the problem of non-interoperability of protocol implementations and develop a tool called PIC (Protocol Interoperability Checker)[1] to identify non-interoperabilities. To our knowledge, PIC is the first tool that addresses this problem.

We say a message *m* exhibits a non-interoperability when certain inputs cause message *m* to be sent by one protocol participant but rejected as non-compliant by the receiver. Message *m* need not be the first message in a protocol interaction; PIC can be used to analyze the *i*-th message (e.g., a data transfer message sent after the connection establishment handshake). Conceptually, PIC characterizes the set of messages that a sender-side implementation can generate, and the associated inputs that generate those messages. Similarly, it characterizes the set of messages a receiver-side implementation considers non-compliant. Messages in the intersection exhibit non-interoperabilities. Thus, unlike manual testing where developers specify test inputs that *may* trigger non-interoperabilities, PIC uses symbolic execution [22] to *automatically* derive test inputs by analyzing protocol code.

A key challenge with this approach is that symbolic execution of protocol code, which tends to be low-level, is invariably incomplete and imprecise, and is hard to scale to real-world implementations. As originally designed, symbolic execution explores as many paths as possible, while we are interested only in exploring paths that produce elements of the sets described above. While we cannot overcome symbolic execution's inherent incompleteness, we address the scaling challenge using two novel techniques.

First, we introduce *joint symbolic execution*, in which the receiver-side symbolic execution is directed based on the sender-side analysis results, dramatically reducing the number of execution paths to consider. Specifically, the only receiver-side paths considered are those that are compatible with messages that can be sent from the sender. As we discuss later, in the absence of joint symbolic execution, an independent analysis may not even be feasible for some protocol implementations. We believe that joint symbolic execution is of general interest for scalable protocol analysis beyond the use case of searching for non-interoperabilities.

[1] PIC is available at `https://github.com/USC-NSL/SPA`

Second, PIC employs new search techniques that direct the analysis toward execution paths more likely to add new messages to the sets being characterized, and therefore to identify interoperability errors. Our experiments show that PIC's search strategies help find 25× more instances of non-interoperabilities than existing strategies within a given time.

We apply PIC to four mature implementations of two qualitatively different protocols: SIP [40] and SPDY [1]. For each protocol, we find thousands of non-interoperable messages spread across different features of the protocol. To understand the causes that lead to these, we group messages that arise from the same underlying problem (e.g., a failure to correctly validate API inputs). For SIP we find 9 distinct causes, and for SPDY we find 13 distinct causes, which fall into several high-level classes: liberal senders, conservative receivers, ambiguous specifications, specifications with optional features and so forth. We reported these to the developers, and several have been fixed.

2 Problem and Background

Protocol implementations cannot easily be validated for compliance against protocol specifications. Such specifications are typically expressed in a natural language and have inherent ambiguities, so ensuring compliance would first require formalizing the specifications in some way, a process which itself can be error prone and open to multiple interpretations. The ambiguity of protocol standards is well-documented, for instance, for BGP [26].

Therefore, instead of compliance to a specification, we focus on checking interoperability of implementations. Specifically, two implementations are considered non-interoperable when there exists at least one protocol message on which they disagree. The disagreement could be about whether the message is protocol compliant (*e.g.,*, the values of header fields are formatted correctly, or take the range of values expected by the receiver) or about what the message means. The work in this paper focuses on the former notion of disagreement. The latter notion of *semantic* interoperability poses additional challenges in extracting meaning and intention from protocol messages, which we leave to future work.

Protocol interactions. To more precisely define our notion of interoperability, consider client-server or peer-to-peer communication between two implementations. Such communication is usually triggered by a call to a protocol API function that implements a specific functionality. This API function initiates a *protocol interaction*: a sequence of message exchanges that perform the desired functionality. For example, a protocol interaction to initiate a call in SIP begins with a client *request* message, followed by a server *response* message. TCP involves several types of protocol interactions: separate API functions initiate connection setup, data transfer, and teardown. In general, a protocol interaction can involve a sequence of messages $m_1, \ldots, m_n$, some going from client to server (or one peer to another), and some the other way. Each message exchange may update the sender or receiver *state*, which consists of the values assigned to variables in the protocol implementation.

Now, consider a message m_i in a protocol interaction. The sender prepares m_i based on the contents of m_{i-1} and on its current state; the first message m_1 is prepared after processing the arguments passed to the API function that initiated the current interaction. At the receiver, m_i is first validated for protocol-compliance. If m_i does not pass these checks, it may be discarded (or an error message sent in response). If it does, subsequent steps of the interaction are invoked.

On the sender side, let S_i denote the set of all possible i-th messages that can be generated (i.e., considered protocol-compliant) given the sender's state after the first $i-1$ messages in the protocol interaction. On the receiver side, let R_i denote the set of all possible i-th messages that the receiver would consider protocol-compliant given its state after the first $i-1$ messages.

Defining non-interoperability. Let $S_i - R_i$ denote the set of messages that are in S_i but not in R_i. Then, the sender and receiver are said to exhibit a non-interoperability if $S_i - R_i \neq \varnothing$, that is, there are messages that the sender can send but the receiver will not deem compliant.

If a sender and receiver are non-interoperable, then we know that the two entities implement the specification inconsistently. However, we cannot directly determine which entity deviates from the protocol specification: non-interoperability can occur either because the sender is too liberal in interpreting the specification, the receiver is too conservative, or *both*.

An example. Figures 1 and 2 list the code for NetCalc, a simple networked calculator protocol that we use to illustrate our approach later. (Functions with `pic_*` names are explained in §3.) At the client, the `Compute()` API function initiates an interaction with the server, sending a message containing an operator and two operands. At the server, the `handleMessage()` function processes the received message. This example has two non-interoperabilities: the client incorrectly expects the server to implement multiplication, and the client does not test for division-by-zero while the server does. So, the set $S_i - R_i$ consists of all messages sent by the client with the multiply operator or messages with zero divisor.

Interoperability testing today. The current practice for testing interoperability of protocol implementations is largely manual and *ad hoc*. Developers of different implementations participate in physical or virtual interoperability events. They test how well two implementations

```
1   Compute(char operator[8], int32 operand1, int32 operand2) {
2     // buffer, buffer size, name
3     pic_api_input(operator, 8, "operator");
4     pic_api_input(operand1, 4, "operand1");
5     pic_api_input(operand2, 4, "operand2");
6
7     byte[] message = new byte[9];
8     if (operator == "plus")
9       message[0] = 0;
10    else if (operator == "minus")
11      message[0] = 1;
12    else if (operator == "divide")
13      message[0] = 2;
14    else if (operator == "multiply")
15      message[0] = 3;
16    else
17      throw exception;
18    message[1..4] = operand1;
19    message[5..8] = operand2;
20
21    // buffer, length, buffer size, name
22    pic_msg_output(message, 9, 9, "message");
23    sendMessage(message);
24  }
25
26  testHarness() {
27    pic_api_entry();
28    char o[8];
29    Compute( o, 0, 0 ); // Dummy arguments
30  }
```

Figure 1: *NetCalc client. The pseudo-code parses API inputs and populates a message before sending it. PIC annotations declare API inputs (l. 3-5), and message outputs (l. 22). A simple test harness initiates a protocol interaction (l. 26-30).*

```
1   handleMessage(byte[] query) {
2     // buffer, buffer size, name
3     pic_msg_input(query, 9, "message");
4
5     int32 operand1 = query[1..4];
6     int32 operand2 = query[5..8];
7     switch (query[0]) {
8       case 0:
9         print(operand1 + operand2);
10        break;
11      case 1:
12        print(operand1 - operand2);
13        break;
14      case 2:
15        if (operand2 == 0)
16          throw exception;
17        print(operand1 / operand2);
18        break;
19      default:
20        throw exception;
21    }
22    pic_valid_path();
23  }
24
25  void testHarness() {
26    pic_msg_handler_entry();
27    handleMessage(new byte[9]);
28  }
```

Figure 2: *NetCalc server. The pseudo-code parses and validates the incoming message and acts on it. PIC annotations are used to declare the message input (l. 3) and the point in the code where the message has been considered valid (l. 22). Invalid messages generate an exception and are detected by not reaching the validity assertion. A simple test harness allocates a message and launches the message handler (l. 25-28).*

interoperate in order to uncover undesirable interactions resulting from code bugs or ambiguities in the standard.

During testing, developers specify and execute a series of interoperability tests on each pair of participating implementations. This is done by configuring those implementations to communicate with each other in a specific manner, by selecting a network topology for the test, and (optionally) by injecting failures or packet losses. Finally, the participants document testing results in event reports [17, 18, 30, 31, 36].

There are two conceptually different components to specifying an interoperability test: which protocol interaction (e.g., connection setup and termination, options negotiation, data transfer, and control commands) to test, and what specific test inputs to use. Specifying the protocol interaction is conceptually straightforward, since there is usually a small number of such features. However, specifying good test inputs (e.g., the call parameters in a SIP connection setup) is difficult. The space of potential protocol inputs can be very large (e.g., all possible URLs). To detect a non-interoperability, the specified test inputs must generate a message in $S_i - R_i$. Today the developer gets essentially no help in this task, resorting to a random search, perhaps guided by intuition, experience, and an understanding of potential ambiguities in the specification. Once a non-interoperability is found, developers converge on a mutually consistent reading of the specification in order to fix it.

Further, given the size of the search space, developers are able to consider only a very small fraction of the possible inputs; as one of the interoperability reports acknowledges, “the test parameters were limited” [18]. As a result, many corner cases go undiscovered during testing and are discovered *after* implementations are deployed in production [35, 39].

This paper explores methods to *derive* test inputs that generate messages in $S_i - R_i$ using program analysis, and instantiates these methods in a tool called PIC. Program analysis simultaneously removes a large burden on developers and improves the effectiveness of interoperability testing via targeted search and increased coverage.

3 PIC Design and Implementation

In this section, we first describe PIC’s high-level approach and then detail its design.

3.1 PIC Approach

Our goal is to develop methods that, with low developer effort, can uncover non-interoperabilities in real-world protocol implementations. We seek methods that are *independent of a specific protocol implementation or type* and can therefore be applied to a wide range of protocols.

Symbolic execution. We use *symbolic execution* [22], a program analysis technique that simulates the execution of code using a symbolic value $\sigma_{\mathtt{x}}$ to represent the value of each variable `x`. As the symbolic executor runs, it updates the symbolic store that maintains information about program variables. For example, after the assignment `y = 2*x` the symbolic executor does not know the exact value of `y` but has learned that $\sigma_{\mathtt{y}} = 2\sigma_{\mathtt{x}}$. At branches, symbolic execution uses a constraint solver to determine the value of the guard expression, given the information in the symbolic store. The symbolic executor then only explores branches when the corresponding boolean guard is satisfiable. For example, it will explore both then and else branches of an if-then-else statement if the condition can be either true or false given the symbolic state of the system. In this way, a tree of possible program execution paths is produced. Each path is summarized by a *path constraint* that is the conjunction of branch choices made to go down that path.

Key insight. In a protocol implementation, the set S_i of messages that can be sent and the set R_i of messages that would be accepted as the i-th message in a protocol interaction *can be succinctly represented by the symbolic values of the message header fields* generated by the sender and accepted by the receiver, when they are both symbolically executed. Thus, $S_i - R_i$ can be computed by determining if there exist concrete values for the message header fields that would match the symbolic constraints on the sender, but not on the receiver (and vice versa).

In theory, we can use an existing symbolic execution tool to compute the sets S_i and R_i, in order to then produce the set $S_i - R_i$ of non-interoperabilities. In practice, however, we face two difficulties. First, symbolic execution is invariably incomplete, since real code can have a large (often unbounded) number of possible executions due to loops, recursion, etc. Thus, what we are bound to get are subsets $S'_i \subset S_i$ and $R'_i \subset R_i$. Using the difference of these subsets $S'_i - R'_i$ to determine whether $S_i - R_i \neq \varnothing$ will lead to many false positives, since many relevant elements of R_i may be missing from R'_i.

To address this limitation, we recast our analysis to ask a question that can be answered precisely even with incomplete sets. Instead of trying to compute R_i, we compute $\neg R_i$, the set of messages that the receiver rejects. Given the limitations of symbolic execution, we will actually obtain some set $\hat{R}_i$ such that $\hat{R}_i \subset \neg R_i$. This results in computing $S'_i \cap \hat{R}_i \subset S_i - R_i$, by which non-interoperabilities are found when $S'_i \cap \hat{R}_i \neq \varnothing$, i.e., there are messages that are generated by the sender but rejected by the receiver. This formulation of interoperability is mathematically identical to the original one when the sets are complete but trades false positives for false negatives in the presence of partial information. This is more useful, since any message that belongs to both S'_i and $\hat{R}_i$ is in fact non-interoperable but does, however, limit PIC's ability to certify interoperability as $S'_i \cap \hat{R}_i = \varnothing \nRightarrow S_i - R_i = \varnothing$.

The second difficulty is that, as originally designed, symbolic execution attempts to cover as many paths as possible, without regard to where they lead. Given the already difficult task of scaling symbolic execution to protocol code, as well as the inherent incompleteness described above, such a blind search is less likely to produce useful results than one directed toward likely non-interoperabilities. Later in this section, we describe our techniques to address this difficulty.

3.2 PIC Architecture

The PIC workflow has four stages, shown in Figure 3. Its input is the intermediate code representation generated from annotated source (*i.e.,* LLVM [2] bitcode).

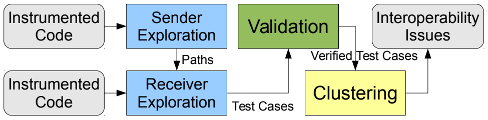


Figure 3: *PIC Analysis Work-Flow*

Annotation	Description
pic_api_entry()	Marks calling function as test harness for API handler.
pic_message_handler_entry()	Marks calling function as test harness for message handler.
pic_api_input(var, size, name)	Annotates buffer as API input.
pic_msg_output(var, length, bufSize, name)	Annotates buffer as message output.
pic_msg_input(var, size, name)	Annotates incoming message buffer of specified size.
pic_msg_input_size(var, name)	Annotates variable specifying size.
pic_valid_path()	Marks code point at which message is considered valid.

Table 1: *PIC Annotations*

Analysis annotations. As with today's interoperability tests, a developer using PIC has to specify (a) the protocol interaction to test for interoperability, (b) what constitutes a non-interoperability, and (c) the API input parameters. These are specified using code annotations (Table 1). In a crucial departure from today's practice, the developer does *not* need to specify concrete values for the test inputs: rather, PIC derives test inputs that trigger non-interoperabilities as described below.

The `pic_api_entry()` and `pic_msg_handler_entry()` annotations identify test harnesses at the client (lines 26-30, Figure 1) and server (lines 25-28, Figure 2), respectively. The `pic_api_input()` annotation (lines 3-5, Figure 1), specifies which inputs are of interest; other inputs are bound to a single concrete value during analysis (limiting its scope to trade completeness for complexity). In Figure 1, these annotations cover all arguments to the client API, but in practice, some of them can be omitted, focusing the analysis only on the specified inputs. Finally, two annotations convey the semantics of non-interoperability. `pic_msg_output()` on the client indicates the codepoint at which a message is transmitted (line 22 in Figure 1). `pic_valid_path()` on the server indicates the codepoint at which a message is considered protocol-compliant (line 22, Figure 2).

Symbolic execution of sender and receiver. The annotated sources are compiled into LLVM bitcode and fed to PIC. The first stage of PIC uses the sender-side annotations to analyze the sender code and generates *paths*, defined by a path constraint and the symbolic value of the resulting message. The second stage *uses these paths*, in a technique we call *joint symbolic execution*, to perform a similar analysis of the receiver, using receiver-side annotations. This results in a set of path constraints that represent non-interoperabilities, which are subsequently passed to a *constraint solver*. The solver determines a satisfying *model*: an assignment of concrete values to symbols that satisfies the formula. As such, the output is a set of concrete instances of non-interoperabilities.

The first two stages use the KLEE [9] symbolic execution engine, modified to use a novel guided search strategy (described below). Being based on KLEE and LLVM, PIC can analyze any language that can be translated to LLVM, which currently includes many popular languages such as C, C++, Objective-C, and C#.

Validation using concrete execution. The third stage removes false positives from the output of the second stage. False positives can arise because the first two phases may produce path constraints that do not in fact represent feasible paths to the target program points. This happens due to the conservative nature of symbolic execution in the face of code that is either not available (e.g., system calls and external libraries) or that cannot be analyzed precisely (e.g., complex heap manipulations).

Therefore, we concretely execute the protocol code, to rule out infeasible paths. The annotations used to declare API inputs are now used to inject concrete input values directly into the running program. At the receiver, whether a validity annotation is reached is now used to confirm or refute the non-interoperability. While a test harness is used to exercise the protocol API during the symbolic execution stage, in the validation stage the application may be used in a setting closer to an actual deployment, including the use of more elaborate network topologies, if needed. The validation stage scales well in our experience: as we show later, it is feasible to validate every produced concrete non-interoperability in our evaluation (some generate 70,000+ non-interoperabilities). n

Clustering. To help developers interpret the results, the final stage clusters the potentially many non-interoperability instances produced post-validation. PIC provides an extensible technique for this purpose.

3.3 Joint Symbolic Execution

Independent symbolic execution, in which S'_i and $\hat{R}_i$ are independently computed, can miss many interoperability bugs, for two reasons. First, the receiver-side analysis will waste a lot of time analyzing messages that the sender would never send, for example because it performs its own validation. Second, without any coordination, the sender and receiver are likely to explore different subsets of the space of possible messages, and by definition interoperability bugs will only surface in the intersection of these subsets.

PIC's joint symbolic execution modifies the receiver-side symbolic execution as follows. Initially, symbolic execution proceeds normally, up until the point where a message is received. The symbolic execution state is then forked multiple ways, one for each sender path. The path constraint on each of these forked states is then manipulated, AND-ing the path constraints from the respective sender path as well as *connecting constraints* that bind the sent message to the received message. Binding is done by declaring a byte-for-byte equality of the received message buffer to the symbolic value established for the message buffer on the sender. Symbolic exploration then proceeds as usual. By construction then, the output of the receiver-side analysis is a set of path constraints that characterize non-interoperabilities (i.e., $S'_i \cap \hat{R}_i$).

Joint symbolic execution has the effect of driving receiver-side symbolic execution only along paths consistent with messages the sender can send. This technique solves the two problems of independent symbolic execution described above. Any non-compliant message that the sender would never send is not explored by the receiver. More importantly, any symbolic message from the sender that represents an interoperability bug will *definitely* be explored by the receiver, without needing to be independently discovered. For example, in NetCalc, if the symbolic executor has only enough resources to produce a path for one of the two non-interoperabilities on each side, independent symbolic execution could produce different ones on each side, with the result that neither interoperability is detected (since the intersection would be empty). For these reasons, as we show in §4, joint symbolic execution is significantly more effective than independent symbolic execution.

3.4 Guiding Symbolic Execution, To and Fro

The original goal of symbolic execution is to explore as many code paths as possible for the purpose of program testing. Real-world protocol implementations have a huge number of paths, and a direct application of symbolic execution can be extremely inefficient.

In our setting, we require a form of guided symbolic execution that preferentially explores paths that satisfy certain properties of interest. Specifically, interoperability testing requires two different kinds of guidance. On the sender side, we need to explore paths that reach one of the few program points where a message is sent, which we call *convergent exploration*. This goal is similar to the notion of *directed symbolic execution* described by Ma *et al.* [28]. On the receiver, we need to explore paths that *avoid* a few points where the message is considered valid, which we call *dispersive exploration*. To our knowledge, there is no prior work on achieving this goal.[2]

Convergent and dispersive exploration. To achieve both convergent and dispersive exploration in a common framework, we reduce the problem of directed symbolic execution to a specialized instance of graph search. Indeed, choosing which paths to further explore is analogous to picking which node to visit next in a graph traver-

[2] One could avoid dispersive exploration by annotating *invalidity* assertions at the receiver and then doing convergent exploration. But that greatly increases the annotation burden, thereby increasing the possibility of erroneous annotations. One of the SPDY implementations we explore requires 2 validity assertions but 28 invalidity assertions.

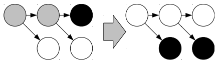


Figure 4: ***Dispersive symbolic execution preprocessing.*** *The black dispersive target on the left is converted to two convergent ones on the right using a reverse control-flow analysis (gray).*

sal. Convergent exploration is a matter of using search strategies for efficiently reaching a target set of program points. This translates into computing a *distance function* that determines an ordering of paths to explore. Different distance functions embody distinct search strategies.

Dispersive exploration, on the other hand, is not so straightforward. A potential approach is to negate the distance function, but that merely guides exploration away from undesirable points. Absent further guidance, this could easily get stuck in endless loops going nowhere. Instead, we conduct an initial pass on the CFG (control flow graph) to map dispersive targets to be avoided into convergent targets to be reached (Figure 4), in effect deriving invalidity assertions from the user specified validity ones. This preprocessing stage starts with a reverse control-flow analysis, finding all points that can reach the dispersive target. Any remaining points cannot reach the dispersive target and are subsequently designated as targets for a convergent exploration.

Fast search. By default, symbolic execution engines use depth-first-search (DFS), which is memory-efficient, but is unaware of the targets. DFS makes early branching decisions, leading to a point deep in the control-flow graph. Once there, it exhaustively explores nearby branches, before backtracking. DFS can waste a lot of time in this localized exploration, or if it finds a non-interoperability, it will find many instances or small variants of the same underlying non-interoperability before discovering a qualitatively different non-interoperability (§4.5).

Ma *et al.* [28] propose three new strategies: call-chain backward symbolic execution (CCBSE), shortest-distance symbolic execution (SDSE), and mixed-chain backward symbolic execution (Mix-CBSE), a hybrid of the first two. CCBSE works backward from the target point, performing (forward) symbolic execution from the nearest enclosing function, and repeats this process backwards until reaching the entry point. This goal-directed approach does not work well when there are a large number of possible targets, as is the case with dispersive exploration, since it requires managing and prioritizing among the several independent analyses for each target.

SDSE, which is similar to the technique in ESD [44], is analogous to greedy best-first search (GreedyBestFS), based on a control-flow distance metric. Unlike CCBSE, it can naturally accommodate multiple targets. But we found that, because of its greediness, it sometimes suffers from stubbornly sticking to potentially bad early branching decisions. This can lead to local minima. An example is the case where one branch of a conditional has a shorter control-flow distance to the target than the other, but may require significantly more symbolic execution (e.g., unraveling loops).

In PIC, we therefore use a strategy based on the A* heuristic search algorithm [41]. A* is a variant of best-first-search that also *considers the cost of the path traversed so far*. This approach allows the search to quickly exit local minima since local exploration increases the path cost, making other paths more attractive. Further, since the control-flow distance metric can be just as easily calculated for many points as for a few, this heuristic permits both convergent and dispersive exploration.

Basic distance heuristic. A good distance heuristic is key to the efficient use of A*. A* does not allow overestimating and significantly under-estimating approximates a breadth-first search, potentially leading to state explosion. The basic distance heuristic that we use is based on a work-list based inter-procedural analysis on the CFG (done before symbolic execution) [12, 37, 38]. In the CFG, nodes represent program points and directed edges connect each node to its possible control-flow successors. Some edges connect two program points in the same function, while others represent function calls and returns. We assign a distance to each node, intuitively representing distance to a target node. For simplicity in the discussion we assume a single target.

Consider a node *n*, that is an ancestor of the target node. If the path from *n* to the target does not traverse a function return, then the distance of *n* is defined to be one more than the minimum distance of any of *n*'s direct successors in the CFG. We call this an *absolute* distance.

However, the distance metric is less clear for program points within functions that are called *and returned from* on the path toward the target, since different call sites to these functions can have very different distances to the target. One could naively work with the minimal distance from any call site to statically compute an absolute distance but, without the context of a call stack, this could significantly underestimate distances during earlier calls. Therefore, for such functions our algorithm computes a *relative* distance for each program point, which is simply the distance to a return point in the function, and we later compute a final distance metric for these nodes on the fly during symbolic execution. Specifically, to compute the distance metric for a node reached during symbolic execution, we traverse the current call stack backward, summing all of the relative distances encountered and stopping when reaching the first absolute distance, which is included in the final sum (Figure 5).

Return normalization. The basic distance metric above works well under most circumstances, but we encountered a problem that occurs frequently in input-parsing

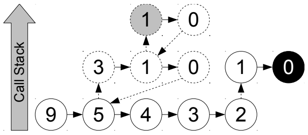


Figure 5: ***Calculating inter-procedural distances:*** *absolute distances from each program point to the black target are calculated on the direct call path (solid circles) and relative distances (dashed circles) are computed to exit points in called functions. The final distance from the gray node is calculated as as the sum of relative distances along the stack up to and including the first absolute one:* $1+1+5=7$.

```
1   int entry( char *input ) {
2     if (checkInput(input) == SUCCESS)
3       target();
4       return SUCCESS;
5     } else {
6       return FAIL;
7     }
8   }
9
10  int checkInput( char *input ) {
11    int i = 0;                        // Distance: 6
12    while (input[i] == ' ')           // Distance: 5
13      i++;                            // Distance: 6
14    if (input[i] != 'g')              // Distance: 4
15      return FAIL;                    // Distance: 3
16    if (input[i+1] != 'o')            // Distance: 3
17      return FAIL;                    // Distance: 2
18    if (input[i+2] != 'o')            // Distance: 2
19      return FAIL;                    // Distance: 1
20    if (input[i+3] != 'd')            // Distance: 1
21      return FAIL;                    // Distance: 0
22
23    return SUCCESS;                   // Distance: 0
24  }
```

Figure 6: *Example showing the importance of equally prioritizing all exits.*

code, where the particular return point from a function affects the ability of the path to eventually reach the target. Figure 6 shows an example. The `entry` function reaches the target and calls function `checkInput` along the way. A naive approach would guide execution within `checkInput` to the nearest return instruction. However, it's clear from the code that only the instruction that returns `SUCCESS` can actually lead to the target point; the other return points signal an error that gets propagated back up the call stack until the message is rejected as invalid.

Augmenting the CFG analysis with the necessary *value sensitivity* to address this problem would require a much more sophisticated and computationally intensive algorithm (akin to symbolic execution itself!). Instead, we modify our metric for return points to prioritize them equally. This will not prevent exploration from taking the wrong ones at some point, but it will mitigate the pathological case in Figure 6 by exploring each return point *before* unraveling the loop on line 12 one more iteration, instead of unraveling all iterations with the first return before trying the next one. This adaptation is achieved by taking into account each exit point's depth, *i.e.,* its minimum control-flow distance from the entry point, in relation to the function's maximum depth. Using this notion, instead of initializing each exit point's distance to 0, each one is assigned a custom distance metric calculated as $maxDepth - depth(node)$, effectively making them all look equidistant (Figure 7). The resulting relative distances are annotated in Figure 6.

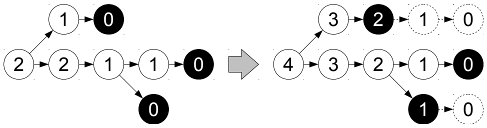


Figure 7: ***Normalizing the costs to exit nodes:*** *the initial cost of exit nodes (black) is adjusted when computing relative distances to avoid favoring any particular one. This is equivalent to adding virtual NOPs (dashed) to early returns so that all return statements appear as if they were at the same depth.*

3.5 State Initialization

PIC provides support for finding non-interoperabilities on a message that occurs somewhere "in the middle" of a protocol. For instance, a developer may want to detect non-interoperabilities for a data transfer, which occurs only after proper connection establishment. For this, PIC provides a lightweight form of message record and replay. During test runs of a protocol, PIC automatically logs the concrete values of messages as well as client and server API inputs at each protocol interaction. Suppose there are m messages in a protocol interaction, and a developer wishes to analyze the k-th message ($k \leq m$). The developer invokes PIC with the log and k as inputs, and PIC begins symbolic execution by feeding in the first $k-1$ messages as concrete values. This sets up the initial state for symbolically executing the k-th message, which proceeds exactly as described above.

This approach enables PIC to symbolically explore the space of k-th messages, predicated on the sets of $k-1$ concrete recorded messages. In future work, we intend to explore scalable iterative joint symbolic execution, exploring all possible combinations of k messages.

3.6 Clustering Non-interoperabilities

During exploration, it is not uncommon for PIC to find many non-interoperable messages or API inputs that stem from the same underlying issue (e.g., improper validation of an API input). It can be difficult for a developer to manually sift through each of these. We implemented an optional analysis step that allows developers to classify the resulting instances into separate clusters.

The developer uses this capability as follows. She first picks one instance and defines a clustering function for it. The function indicates whether a given instance belongs to the cluster. It can use as input any attribute of the instance, such as the protocol API that was called, conditions on the API input, or even a regular expression on the message content. Then, she runs the clustering function(s) defined thus far, at the end of which there are several unclassified test cases. She picks one of these, and repeats the steps above until no unclassified test cases remain. In one of our evaluations, PIC produced over 17,000 non-interoperability instances, which were eventually clustered into 6 or 7 groups. The developer effort is proportional to the number of clusters, not the number of instances and, from our experience, is not significant. The most elaborate such function we developed copied a state-machine from the original code and detected a particular transition. With such an iterative process, clustering becomes intertwined with debugging and can be seen as a means of filtering instances of non-interoperabilities that have already been classified. We have left automating this process to future work.

3.7 Stage Pipelining and Parallelism

Despite our optimizations, analyzing real protocol implementations can take a significant amount of time. Our implementation uses pipelining between the sender-side and receiver-side analyses and between the receiver-side and the validation stages. It also uses parallelism within the receiver-side analysis stage and the validation stage. We omit the details of these optimizations for brevity.

4 Evaluation

To evaluate PIC, we selected two different protocols to uncover non-interoperabilities: the Session Initiation Protocol (SIP), a signaling protocol for Internet telephony systems; and SPDY, a widely-deployed protocol for accelerating Web transfers. SIP was chosen due to its prior history of interoperability problems [35, 39], and SPDY was chosen because it is a recently developed, but rapidly evolving, protocol that is already implemented in the latest versions of browsers and servers and deployed on many major content providers. There is one additional important difference between the two protocols: whereas SIP headers are human readable text-based messages, SPDY is a binary protocol where messages have a more rigid structure. That PIC is able to analyze both classes of protocols demonstrates its generality.

We analyzed two SIP implementations (eXoSIP and PJSIP), and two versions each of two SPDY implementations (spdylay and nginx), using the following procedure. We first defined use cases, defining the client-server roles and assumptions regarding network state (*e.g.,* a SPDY client fetching content from a server on localhost). We then created simple test harnesses that essentially exercised a concrete example of a protocol interaction matching the selected use case (*e.g.,* invoking the SPDY request API with a partially symbolic URL in the form of "http://127.0.0.1/*"). Using a debugger, we then followed the code as it processed API and message inputs in order to determine where the message was sent and received and where the message passes validation and begins to be handled. These steps helped determine where to annotate the code. When analyzing later messages in an interaction, we used the annotations to record and replay inputs and messages from a test run, before initiating symbolic execution. After this, the analysis proceeded as described in §3.1.

Implementation	Library	Annotations	Test Harness
eXoSIP-3.6.0	43990	38	209
PJSIP-1.12	83916	21	-
spdylay-0.3.7	22739	23	420
spdylay-1.3.1	25495	23	420
nginx-1.5.5	99446	4	-
nginx-1.7.4	104364	4	-

Table 2: *Source lines of code for library, annotations and test harnesses.*

Table 2 shows that PIC scales to implementations that involve tens of thousands of SLoC. Moreover, we see that the effort of adding the annotations is small, especially compared to the size of the original code base. For example, for nginx, since we analyzed only one message, only 4 annotations were needed. For PJSIP and nginx, a separate test harness was not needed as we could simply re-use its command-line application. As an aside, although PIC was designed for protocol developers, we (the authors) did not develop any of the analyzed code; that we were able to find significant non-interoperabilities in these large implementations is an example of how automating the search for test inputs can reduce reliance on developer insight and intuition.

4.1 SPDY results

We evaluate two implementations of SPDY. The first is a modular SPDY stack called spdylay [42], for which we analyze versions v0.3.7 and v1.3.1. spdylay offers both client and server functionality. The second implementation is spdylay, a popular open-source Web server, for which we analyze versions v1.5.5 and v1.7.4. The former version supports only SPDY v2, but the latter includes support for v3 and v3.1.

Our analyses explore all 4 client-server combinations across these versions. Certain components of SPDY (data encryption and compression) are challenging for symbolic execution, since analyzing them can be analogous to reversing one-way functions. State-of-the-art techniques usually treat these as uninterpreted functions and then use developer-supplied invariants (*e.g.,* $Decrypt(Encrypt(data,key),key) = data$) to simplify the resulting expressions (*e.g.,* cancel out the encryption once the client has decrypted the cypher-text with the same key). Since KLEE doesn't support this kind of evaluation, we abstracted these into identity functions during analysis (but not during validation). This of course limits PIC's ability to detect non-interoperabilities that originate in these functions, but was necessary to enable analysis of the remaining protocol code.

Our experiments focused on the Stream Creation (SYN_STREAM) message and the Stream Response (SYN_REPLY) message (*the second message in the protocol interaction*). The non-interoperabilities discovered for the latter message are a subset of those for the former, so we focus on describing results for the former.

The results are summarized in Table 3. To obtain

#		spdylay 0.3.7	spdylay 0.3.7	spdylay 1.3.1	spdylay 1.3.1		
	Inputs	spdylay 0.3.7	nginx 1.5.5	spdylay 1.3.1	nginx 1.7.4	Cause	Verdict
A	hName="\x1"	✘				The client insufficiently validates API input, allowing control characters in header names. The server checks for and rejects these characters. The client API should fail if illegal characters are found.	Liberal Sender
B	hValue1="\x1"			✘		The spdylay server now validates header characters more carefully. The new validation was not applied on the client.	
C	hValue1=""		✘	✘		Empty header values still seems to escape client-side validation.	
D	path="/%%"		✘		✘	spdylay doesn't percent encode illegal characters.	
E	version= "HTTP/0.9"		✘		✘	nginx rejects any HTTP version prior to 1.0 if used in SPDY.	Conservative Receiver
F	hValue1="\n"		✘		✘	nginx doesn't allow either CR or LF characters in header values.	
G	path="/.."		✘		✘	nginx carefully parses the path hierarchy and prevents breaking out from the web root.	
H	hName=":"				✘	nginx only allows the the ":method", ":path", ":version", ":host", and ":scheme" headers to start with ":".	
J	hValue1="" hValue2=""	✘				In some circumstances, the client allows empty header values, whereas the server doesn't. An "off-by-one" error masks the check on the server when only the last header value is empty. The server is otherwise correct in disallowing empty header values, but should also check the last header. The client API should fail if empty values are found.	Bug
K	path="/%00"		✘			nginx specifically rejects the correctly encoded NUL character in the path.	Optional Feature
L	method= "TRACE"		✘			nginx doesn't allow the TRACE HTTP method.	
M	cliVersion=3		✘			This version of nginx doesn't yet support SPDY/3 and fails to parse messages. It also drops the message instead of responding with the specified error.	Unsupported Version
N	cliVersion=2 srvVersion=3	✘		✘		In the spdylay API the server application must specify the protocol version (usually negotiated by SSL). Messages processed in the wrong context are invalid. The version specified in incoming messages should be used instead.	

Table 3: *Interoperability issues in SPDY.*

these results, we constrained the PIC analysis to explore a subset of the protocol inputs for SPDY, namely the client version or `cliVersion`, the server version `srvVersion`, header names and values `hName` and `hValue`, and the HTTP `path`, `method`, and `version`. PIC found several tens of thousands of instances of non-interoperabilities, which we classify into 12 clusters; for brevity, we omit exact counts per cluster. A cross indicates that the cluster manifested in the corresponding client-server combination.

To gain insight into the underlying issues that cause non-interoperabilities, based on our reading of the specification, we classified the non-interoperability clusters into 5 qualitatively different sub-categories, discussed below. Because PIC has a general definition of non-interoperability (§2), it can discover non-interoperabilities that stem from many different underlying issues.

Liberal sender. A long-standing guideline for protocol developers has always been: be conservative in what you send, and liberal in what you receive. Non-interoperabilities can arise when senders are more liberal than receivers. For example, the spdylay client permits control characters in header names, but the spdylay server does not (**A** in Table 3). We reported this error to the spdylay developer, who fixed it in the newer version of spdylay. However, in fixing this error, spdylay added newer code to validate control characters in headers and values, but the developer appears to have forgotten to validate values for control characters on the client side, introducing a new non-interoperability where there was none previously (**B**). When we contacted the developer about this new bug, he was hesitant to fix it because he thought clients using the spdylay library may already be leveraging the library's lax checking of control characters; in effect, the developer seems inclined to preserve "bug compatibility." This fear of breaking compatibility once non-interoperability has been released into the wild motivates systematic checking prior to the release.

Similarly, the spdylay client is liberal in permitting empty header values, while the server rejects requests with empty headers (**C**). In analyzing the test cases for this non-interoperability, and discussing them with the spdylay developer, we discovered an implementation error, which we describe below. Finally, the spdylay client does not escape non-ASCII characters correctly in the path, which the nginx server appropriately rejects (**D**).

Implementation error. This non-interoperability (**J**) between spdylay client and server in the older version is more subtle and is unlikely to have been found by manually designed test inputs. The spdylay client allows empty values in name-value pairs, and the server usually checks for these and correctly skips them except in one corner case. The assumption that the beginning of a header value cannot happen at the last position of the decompressed packet payload masks the check, in what looks like an "off-by-one" error. The spdylay developer fixed it in the newer version of spdylay.

Conservative receiver. Non-interoperability can also arise when receivers are more conservative than what the specification requires, either because the specification is ambiguous, or for security reasons. One such non-interoperability is between spdylay and nginx. Although the nginx web server supports HTTP v0.9, it disallows tunneling HTTP 0.9 over SPDY (**E**). This non-interoperability is subtle because it occurs within a tunneled protocol, and demonstrates the power of the kind of systematic analysis that PIC performs. The SPDY specification does not require servers to prevent HTTP 0.9 tunneling within SPDY: the nginx developers appear to have made an undocumented assumption that clients are unlikely to be using SPDY to tunnel HTTP 0.9. While this may be true, it is another instance of the benefit of systematic analysis to uncover such undocumented assumptions.

A second non-interoperability in this category occurs because nginx prevents paths that traverse up the directory hierarchy using "/.." (**G**): this is a security feature designed to prevent attackers from breaking out of the web root and accessing files from elsewhere in the filesystem hierarchy. The nginx code for determining this involves a fairly sophisticated state machine and PIC was able to symbolically traverse the state machine to uncover the non-interoperability.

A third non-interoperability in this category occurs with SPDY v3 on nginx. This version of SPDY merges normal HTTP headers with new ones added for tunneling, requiring tunneling headers to be prefixed with ':' (**H**). The specification is silent on whether other headers may be colon-prefixed, and nginx conservatively only permits a specific set of headers to be colon-prefixed.

Finally, nginx conservatively rejects header values containing carriage returns and linefeeds; the SPDY specification is silent on this point (**F**).

Optional features. Non-interoperabilities can also occur because specifications permit optional features. One example in this category is that a spdylay client can generate a SPDY request with an HTTP TRACE method (a valid HTTP method defined in the spec), which nginx does not support in either of its versions because of a cross-site scripting vulnerability (**L**). A second non-interoperability in this category is that spdylay permits generation of URL paths with a NUL character (escaped using '%', **K**). The specification for URI generation permits receivers to be conservative and reject paths with NUL characters in them; however, URIs with arbitrary binary characters (including the NUL character) are used in RESTful APIs, so some Web servers permit them.

Unsupported versions. The first non-interoperability in this category occurs between a spdylay client and server. Spdylay permits server applications to specify a version number, and incoming messages from clients are processed in that context, even if a client specified a different version number (**M**). While this is a relatively obvious error, with a simple fix—ensure that incoming messages are processed using the embedded version number—it was still surprising to see this error in a stack against which a client, a server, and a proxy have been developed. In discussions with the developer, it became clear there is an undocumented assumption that spdylay will be used with SSL, which negotiates the protocol version out-of-band. PIC, being a systematic tool, can unearth such undocumented assumptions.

A second non-interoperability, between the older spdylay and the older nginx, occurs because the older nginx does not support SPDY v3 (**N**). While this doesn't match a colloquial notion of interoperability, it matches our definition: spdylay generates a v3 request, but nginx rejects that message. As expected, this disappears in the newer nginx which supports SPDY v3.

Discussion. Several of the spdylay non-interoperabilities have been fixed. We have communicated the nginx non-interoperabilities to the developers and are awaiting their feedback. To quantify the complexity of these non-interoperabilities, we counted the number of path constraints minus the connecting constraints (§3.3) in the results reported by PIC: all of these non-interoperabilities contained *between 60 and 80 path constraints*. Roughly speaking, a blind search for these non-interoperabilities would have required searching a space of at least 2^{60} message-header combinations. Developer intuition can likely reduce this search space, but that alone is not sufficient. That is why we find many non-interoperabilities in our analysis, even when the client and server code was developed by the same developer (spdylay).

4.2 Session Initiation Protocol (SIP) results

SIP includes several messages for features such as establishing, answering, forwarding, and terminating calls; sending instant messages; and subscribing to events (*e.g.*, user presence). For our analysis we chose two mature and well-known SIP stacks: eXoSIP, an extension of the GNU oSIP library, as sender and receiver, and PJSIP, as a receiver. Our experiments with SIP implementations bring out two capabilities of PIC not highlighted above: PIC can be used to analyze later messages in a protocol interaction, and to iteratively uncover non-interoperabilities that reveal themselves only after existing non-interoperabilities are fixed.

Table 4 shows the 9 clusters of non-interoperabilities generated while running eXoSIP as client and PJSIP as server. We also ran eXoSIP as client and server, which does not exhibit any of these non-interoperabilities.

The discovered non-interoperabilities span different message types: call establishment (INVITE), feature discovery (OPTIONS) and event subscription (SUBSCRIBE). For each of these messages, we analyzed interoperability for three headers: `from`, `to`, and `event`.

Most of the SIP non-interoperabilities fall into a *liberal sender* category. The eXoSIP sender permits control characters in the `from` (**P**, **S**), `to` (**Q**, **T**) and `event` fields (**W**), which the PJSIP server rejects. The eXoSIP sender also permits `to` header inputs where the URI schema is confused with other parts of the URI (such as the display name); PJSIP rejects these malformed URIs (**Q**). The eXoSIP developer acknowledged these non-interoperabilities, but pointed out that eXoSIP is a library that performs minimal input validation. However, this appears to place an undue burden on the developer who

Message	#	Inputs	Cause	Verdict
INVITE (Call Initiation)	P	from="<\x2@127.0.0.1:5061>"	eXoSIP insufficiently validates API input, allowing control characters in the "from" field. PJSIP is unable to parse the resulting URI. The application should prevent such chars from being used.	Liberal Sender
	Q	to="sip\":c@127.0.0.1:5060>"	The eXoSIP parser seems to confuse the SIP URI scheme with other parts of the URI (e.g., the display-name). PJSIP is unable to parse the resulting URI. A robust scheme detection mechanism in the application could prevent such issues.	
	R	from="\"\"<si@127.0.0.1:5061>"	eXoSIP seems to allow the SIP URI scheme ("sip:") to be omitted, whereas PJSIP requires it. The server is unable to parse the resulting URI. The application should check for the absence of the scheme and add one when necessary.	
OPTIONS (Feature Discovery)	S	from="\"\"\x2<@127.0.0.1:5061>"	eXoSIP reuses code for the "from" field in both OPTIONS and INVITE messages. Invalid characters are still not checked for.	
	T	to="<SIP\":@127.0.0.1:5060>"	eXoSIP reuses code for the "to" field in both OPTIONS and INVITE messages. The URI scheme is still misinterpreted.	
	U	from="@127.0.0.1:5061>"	eXoSIP reuses code for the "from" field in both OPTIONS and INVITE messages. The URI scheme is still optional.	
OPTIONS Response (Feature Discovery)	V	header=" "	eXoSIP affords significant flexibility in generating new headers in response messages, but doesn't validate application input on the header name, allowing control characters to mangle the message. PJSIP is unable to parse the resulting message. The application should be careful not to inject syntactically incorrect headers.	
SUBSCRIBE (Event Subscription)	W	event="\r\n"	eXoSIP doesn't validate application input on the event header, allowing control characters to mangle the message. PJSIP is unable to parse the resulting message. The eXoSIP API should fail if illegal characters are found.	
	X	event="aaa"	After implementing simple character checking semantics, PIC creates valid tokens. PJSIP, however, implements more advanced event semantics and only allows subscription to known event types.	Optional Feature

Table 4: *Interoperability issues in SIP with spdylay as client and spdylay as server.*

uses the eXoSIP library to understand the details of the protocol standard. Furthermore, eXoSIP validates some inputs but not others, with no documented guidance for developers on what input validation is left to the application and what eXoSIP performs. In these circumstances, PIC can be used by library or application developers to discover the types of input validation that need to be performed at the application level to avoid non-interoperability issues when used with another implementation in production. eXoSIP also exhibits another liberal sender non-interoperability, omitting the "sip:" URI scheme (**R**, **U**). This truncated URI is permitted by eXoSIP, so it appears to be an undocumented assumption that permits a deviation from the standard for the case when eXoSIP clients talk to eXoSIP servers.

Later messages in a protocol interaction. We used PIC to analyze the OPTIONS response; *this experiment exercised PIC's ability to analyze deeper messages (§3.5).* In our experiment a valid OPTIONS request was generated in PJSIP and replayed into eXoSIP prior to analysis. When eXoSIP receives an OPTIONS request, it returns an OPTIONS response with a set of features specified by the application; however, as a liberal sender, eXoSIP does not validate this application input (**V**).

Iterative testing for non-interoperabilities. SIP allows applications to subscribe for specific "events" (e.g., status changes in a buddy list). The eXoSIP stack permits control characters in event names, which PJSIP rejects. We then fixed this non-interoperability in eXoSIP, and re-ran PIC. This time, PIC generated *another* non-interoperability caused by the fact that although now eXoSIP generates syntactically correct event tokens, it does not check if these match valid event tokens that are known to the remote end (**X**). In this case, the PJSIP server returns an error in response to the SUBSCRIBE message; correctly so, since supporting new client-defined features is an *optional feature* in the protocol. Given enough time, analyzing the unpatched eXoSIP would have found this issue. However, applying fixes such as this one can accelerate the process of finding deeper issues.

4.3 Joint vs. Independent Symbolic Execution

We compared joint and independent symbolic execution on our protocol implementations. On the newer versions of spdylay (as client) and nginx (as server), we find that *joint symbolic execution produces over 100,000 paths in half a day, while independent symbolic execution produces none in the same time.* This performance gap rendered independent symbolic execution completely ineffective for these implementations. On SIP and the older version of spdylay (client and server), independent symbolic execution is able to find the non-interoperabilities, but is slower than joint symbolic execution by orders of magnitude. There appears to be a performance cliff for independent symbolic execution with the newer versions of spdylay and with nginx, which support multiple SPDY versions and are more complex implementations. Beyond this cliff, independent symbolic execution cannot be used, and joint symbolic execution is needed.

4.4 Impact of Search Strategy

Figure 8 compares three search strategies from §3.4: depth-first-search (DFS), best-first-search (GreedyBestFS, which is equivalent to SDSE from [28]), and our customized A*. It also evaluates A* without the return normalization heuristic. Without a reference implementation to work with, we are unable to compare with CCBSE and Mix-CCBSE. The results in Figure 8 indicate a clear performance advantage of A* with return normalization: within an hour, it discovers 25× more test inputs than state-of-the-art approaches. Further, within this time, A* found 5 of the 8 clusters of non-interoperabilities affecting the older nginx, while both A* without return normalization and GreedyBestFS detect only 3, and DFS none.

For this particular scenario, A* without return normalization performed as well as GreedyBestFS. This suggests that, for these implementations, early returns represent the most important cause of local minima. In other

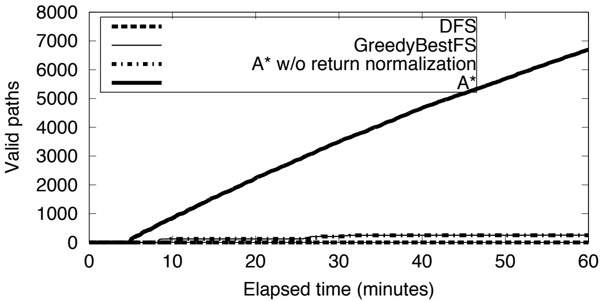


Figure 8: *Performance analysis of four search strategies: DFS, GreedyBestFS, and A* with and without return normalization. The plot illustrates the number of non-interoperabilities produced over time, while analyzing spdylay with nginx. The trials were performed on 10 servers totaling 216 CPU cores.*

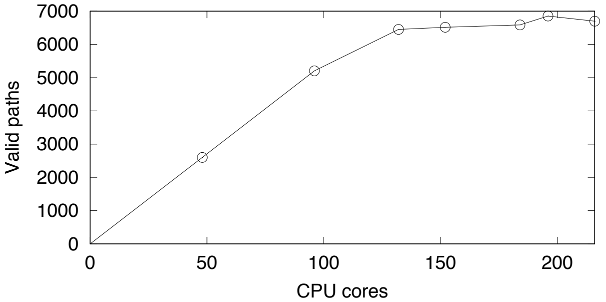


Figure 9: *Scaling analysis showing the number of non-interoperabilities found after 1 hour of analyzing spdylay with nginx, while varying the number of cores.*

settings, we believe A*'s local minima avoidance could outperform GreedyBestFS, even without return normalization.

4.5 Performance Micro-benchmark

To micro-benchmark PIC's performance, we measure the computation time for the main analysis stages, in isolation, for the newer spdylay against the newer nginx. This setup represents the most complex protocol combination we have tested to date. The results show that the client-side analysis took just over 5 hours to generate 60,000 paths, while the server took a little over half a day to generate more than twice that number.

We also ran a scaling analysis in Figure 9. The server-side analysis is pleasantly parallelizable as each client-path is independent but the client-side uses one core and bottlenecks the server at around 132 cores. Scaling out symbolic execution has been done [8], but we leave integrating such an approach for future work.

5 Related Work

Protocol analysis: Prior work has used high-level specifications, using finite state machines, higher-order logic [3, 4], or domain-specific languages [19] to perform a formal verification of protocols. While such specifications are powerful tools to reason about protocol behavior, they do not ensure correctness of implementation. PIC focuses on protocol implementations rather than manually derived formal specifications. Researchers have also explored the use of explicit-state model checkers to find bugs in protocol implementations [20, 21, 27, 32, 33, 43]. To our knowledge, model checking has not been used to discover non-interoperabilities. Like model checking, PIC faces similar challenges in path explosion and uses execution steering techniques to scale the analysis.

Symbolic execution: Godefroid *et al.* [15] and Cadar and Engler [10] developed a general technique for combining symbolic execution with concrete execution to generate test inputs. Since then, researchers have built mature tools using this approach [9, 16], have used it for finding program errors [6, 11], and have enhanced the basic technique in various ways [14, 28, 29]. Others have focused on making symbolic execution more efficient either by directing the search process [13, 28, 34, 44], or by merging states to reduce the search space [24]. We have compared PIC's approach to existing directed symbolic execution techniques (§4.4). State merging, on the other hand, is an orthogonal approach that could potentially be useful in merging paths exchanged during joint symbolic execution, and we have left to future work an exploration of this technique. Further, researchers have used symbolic execution to analyze protocols, but for properties other than interoperability: to determine *equivalence* between two implementations playing the role of the same network service [5, 25], to uncover manipulation attack vectors [23], and to discover bugs in layered server implementations [7]. While this body of work uses tools similar to those used by PIC, it focuses on fundamentally different problems. PIC's focus on interoperability between senders and receivers is unique to the best of our knowledge and motivates new techniques.

6 Conclusion

We presented PIC, which discovers interoperability problems in real protocol implementations. It uses program analysis to infer the sets of messages that one implementation can send but the other rejects. To scale the analysis, it uses joint symbolic execution, in which the receiver-side analysis is seeded by results from the sender. This technique was crucial for PIC and may be generally useful for analyzing interacting protocol implementations. On mature implementations of two protocols, PIC found thousands of instances of non-interoperabilities, across multiple message types and fault causes. Many of the issues have been acknowledged as undesirable by developers and some have already been fixed.

Acknowledgements: We thank the NSDI reviewers and our shepherd Petros Maniatis for helpful feedback on this paper. This work is supported in part by the US National Science Foundation (grant CNS-1161595) and by Cisco Systems.

References

[1] SPDY Protocol - Draft 3.1. http://www.chromium.org/spdy/spdy-protocol/spdy-protocol-draft3-1.

[2] The LLVM Compiler Infrastructure Project. http://llvm.org/.

[3] BHARGAVAN, K., OBRADOVIC, D., AND GUNTER, C. A. Formal verification of standards for distance vector routing protocols. *J. ACM 49*, 4 (July 2002).

[4] BISHOP, S., FAIRBAIRN, M., NORRISH, M., SEWELL, P., SMITH, M., AND WANSBROUGH, K. Rigorous specification and conformance testing techniques for network protocols, as applied to TCP, UDP, and sockets. SIGCOMM '05.

[5] BRUMLEY, D., CABALLERO, J., LIANG, Z., NEWSOME, J., AND SONG, D. Towards automatic discovery of deviations in binary implementations with applications to error detection and fingerprint generation. SS'07.

[6] BRUMLEY, D., POOSANKAM, P., SONG, D. X., AND ZHENG, J. Automatic patch-based exploit generation is possible: Techniques and implications. In *Security and Privacy* (2008).

[7] BUCUR, S., KINDER, J., AND CANDEA, G. Making Automated Testing of Cloud Applications an Integral Component of PaaS. In *APSys 2013*.

[8] BUCUR, S., URECHE, V., ZAMFIR, C., AND CANDEA, G. Parallel symbolic execution for automated real-world software testing. EuroSys '11.

[9] CADAR, C., DUNBAR, D., AND ENGLER, D. Klee: unassisted and automatic generation of high-coverage tests for complex systems programs. OSDI'08.

[10] CADAR, C., AND ENGLER, D. R. Execution generated test cases: How to make systems code crash itself. In *SPIN* (2005), vol. 3639, Springer.

[11] CADAR, C., TWOHEY, P., GANESH, V., AND ENGLER, D. Exe: A system for automatically generating inputs of death using symbolic execution. In *CCS* (2006).

[12] FÄHNDRICH, M., REHOF, J., AND DAS, M. Scalable context-sensitive flow analysis using instantiation constraints. In *PLDI 2000*.

[13] GE, X., TANEJA, K., XIE, T., AND TILLMANN, N. DyTa: Dynamic symbolic execution guided with static verication results. In *ICSE 2011, Demonstration*.

[14] GODEFROID, P. Compositional dynamic test generation. POPL '07.

[15] GODEFROID, P., KLARLUND, N., AND SEN, K. DART: directed automated random testing. In *PLDI* (2005).

[16] GODEFROID, P., LEVIN, M. Y., AND MOLNAR, D. Automated whitebox fuzz testing. In *NDSS 2008*.

[17] GUNASINGHE, H. SCIM interop event at IETF 83rd meeting. http://hasini-gunasinghe.blogspot.com/2012/03/scim-interop-event-at-ietf-83rd-meeting.html, Mar. 2012.

[18] HALEPLIDIS, E., OGAWA, K., WANG, W., AND SALIM, J. H. Implementation report for forwarding and control element separation (ForCES). RFC 6053 http://tools.ietf.org/html/rfc6053, Nov. 2010.

[19] Information processing systems – Open Systems Interconnection – LOTOS – A formal description technique based on the temporal ordering of observational behaviour, 1989.

[20] KILLIAN, C., ANDERSON, J. W., JHALA, R., AND VAHDAT, A. Life, death, and the critical transition: finding liveness bugs in systems code. NSDI'07.

[21] KILLIAN, C. E., ANDERSON, J. W., BRAUD, R., JHALA, R., AND VAHDAT, A. M. Mace: language support for building distributed systems. PLDI '07.

[22] KING, J. C. Symbolic execution and program testing. *CACM 19*, 7 (1976).

[23] KOTHARI, N., MAHAJAN, R., MILLSTEIN, T., GOVINDAN, R., AND MUSUVATHI, M. Finding Protocol Manipulation Attacks. In *SNAP* (August 2011).

[24] KUZNETSOV, V., KINDER, J., BUCUR, S., AND CANDEA, G. Efficient state merging in symbolic execution. In *PLDI 2012*.

[25] KUZNIAR, M., PERESINI, P., CANINI, M., VENZANO, D., AND KOSTIC, D. A SOFT way for OpenFlow switch interoperability testing. In *CoNEXT* (2012).

[26] LABOVITZ, C., AHUJA, A., ABOSE, A., AND JAHANIAN, F. An Experimental Study of Delayed Internet Routing Convergence. In *SIGCOMM 2000*.

[27] LEE, H., SEIBERT, J., KILLIAN, C., AND NITA-ROTARU, C. Gatling: Automatic attack discovery in large-scale distributed systems. NDSS 2012.

[28] MA, K.-K., KHOO, Y. P., FOSTER, J. S., AND HICKS, M. Directed symbolic execution. In *SAS* (September 2011), vol. 6887 of *Lecture Notes in Computer Science*.

[29] MAJUMDAR, R., AND XU, R.-G. Directed test generation using symbolic grammars. ASE '07.

[30] MASINTER, L. WebDAV interop report. http://www.webdav.org/users/masinter/interop/report.html, July 1999.

[31] MOME interoperability testing event. http://www.ist-mome.org/events/interop/, July 2005.

[32] MUSUVATHI, M., AND ENGLER, D. R. Model checking large network protocol implementations. NSDI'04.

[33] MUSUVATHI, M., PARK, D. Y. W., CHOU, A., ENGLER, D. R., AND DILL, D. L. Cmc: a pragmatic approach to model checking real code. *SIGOPS Oper. Syst. Rev. 36*, SI (Dec. 2002).

[34] PERSON, S., YANG, G., RUNGTA, N., AND KHURSHID, S. Directed incremental symbolic execution. In *PLDI 2011*.

[35] RAO, A., AND SCHULZRINNE, H. Real-world SIP interoperability: Still an elusive quest. http://www.sipforum.org/component/option,com_docman/task,doc_view/gid,124/, 2007.

[36] RCS VoLTE interoperability event 2012. http://www.msforum.org/interoperability/RCSVoLTE.shtml, Oct. 2012.

[37] REHOF, J., AND FÄHNDRICH, M. Type-base flow analysis: from polymorphic subtyping to cfl-reachability. In *POPL 2001*.

[38] REPS, T. W. Program analysis via graph reachability. In *International Symposium on Logic Programming*.

[39] ROSENBERG, J. Basic level of interoperability for session initiation protocol (SIP) services (BLISS) problem statement. Internet draft http://tools.ietf.org/html/draft-ietf-bliss-problem-statement-04, Mar. 2009.

[40] ROSENBERG, J., SCHULZRINNE, H., CAMARILLO, G., JOHNSTON, A., PETERSON, J., SPARKS, R., HANDLEY, M., AND SCHOOLER, E. SIP: Session Initiation Protocol. RFC 3261, June 2002.

[41] RUSSELL, S. J., AND NORVIG, P. *Artificial Intelligence - A Modern Approach (Third Edition)*. Pearson Education, 2010.

[42] TSUJIKAWA, T. Spdylay - SPDY C Library. http://spdylay.sourceforge.net/.

[43] YABANDEH, M., KNEZEVIC, N., KOSTIC, D., AND KUNCAK, V. Crystalball: Predicting and preventing inconsistencies in deployed distributed systems. NSDI '09.

[44] ZAMFIR, C., AND CANDEA, G. Execution synthesis: A technique for automated software debugging. In *EuroSys 2010*.

Checking Beliefs in Dynamic Networks

Nuno P. Lopes
Microsoft Research

Nikolaj Bjørner
Microsoft Research

Patrice Godefroid
Microsoft Research

Karthick Jayaraman
Microsoft Azure

George Varghese
Microsoft Research

Abstract Network Verification is a form of model checking in which a model of the network is checked for properties stated using a specification language. Existing network verification tools lack a *general specification language* and *hardcode* the network model. Hence they cannot, for example, model policies at a high level of abstraction. Neither can they model dynamic networks; even a simple packet format change requires changes to internals. Standard verification tools (e.g., model checkers) have expressive specification and modeling languages but do not scale to large header spaces. We introduce Network Optimized Datalog (NoD) as a tool for network verification in which both the specification language and modeling languages are Datalog. NoD can also scale to large to large header spaces because of a new filter-project operator and a symbolic header representation.

As a consequence, NoD allows checking for beliefs about network reachability policies in dynamic networks. A belief is a high-level invariant (e.g., “Internal controllers cannot be accessed from the Internet”) that a network operator *thinks* is true. Beliefs may not hold, but checking them can uncover bugs or policy exceptions with little manual effort. Refuted beliefs can be used as a basis for revised beliefs. Further, in real networks, machines are added and links fail; on a longer term, packet formats and even forwarding behaviors can change, enabled by OpenFlow and P4. NoD allows the analyst to model such dynamic networks by adding new Datalog rules.

For a large Singapore data center with 820K rules, NoD checks if any guest VM can access any controller (the equivalent of 5K specific reachability invariants) in 12 minutes. NoD checks for loops in an experimental SWAN backbone network with new headers in a fraction of a second. NoD generalizes a specialized system, SecGuru, we currently use in production to catch hundreds of configuration bugs a year. NoD has been released as part of the publicly available Z3 SMT solver.

1 Introduction

> Manually discovering any significant number of rules a system must satisfy is a dispiriting adventure — *Engler et al. [13].*

Our goal is to catch as many latent bugs as possible by static inspection of router forwarding tables and ACLs without waiting for the bugs to trigger expensive live site incidents. In our operational network, we see roughly one customer visible operational outage of more than one hour every quarter across our major properties; these live site incidents are expensive to troubleshoot and reduce revenue and customer satisfaction. As businesses deploy services, bug finding using static verification will become increasingly essential in a competitive world.

We have already deployed an early version of our checker in our public cloud; it has regularly found bugs (§ 7). Operators find our checker to be indispensable especially when rapidly building out new clusters. However, we wish to go deeper, and design a more useful verification engine that tackles two well-known [2, 15] obstacles to network verification at scale.

O1. Lack of knowledge: As Engler and others have pointed out [12, 14], a major impediment is determining what specification to check. Reachability policies can be thought of intuitively as “who reaches who”. These policies evolve organically and are in the minds of network operators [2], some of whom leave. How can one use existing network verification techniques [1, 22–24, 26, 35] when one does not know the pairs of stations and headers that are allowed to communicate?

O2. Network Churn: Existing network verification techniques assume the network is static and operate on a static snapshot of the forwarding state. But in our experience many bugs occur in buildout when the network is first being rolled out. Another particularly insidious set of bugs only gets triggered when failures occur; for example, a failure could trigger using a backup router that is not configured with the right drop rules. Beyond such short term dynamism, the constant need for cost reduction and the availability of new mechanisms like SDNs keeps resulting in new packet formats and forwarding behaviors. For example, in recent years, we have added VXLAN [25] and transitioned to software load balancers in our VM Switches [30]. The high-level point is that verification tools must be capable of modeling such dynamism and provide insight into the effect of such changes. However, all existing tools we know of including Anteater [26], VeriFlow [24], Hassel [23] and NetPlumber [22] assume fixed forwarding rules and fixed packet headers, with little or no ability to model faults or even header changes.

Our approach to tackling these obstacles is twofold, and is embodied in a new engine called Network Optimized Datalog (NoD).

A1. General specification language to specify beliefs: We use Datalog to specify properties. Datalog is far more general than the regular expression language used in NetPlumber [22], the only existing work to provide a specification language. For example, Datalog allows specifying differential reachability properties across load balanced paths. More importantly, Datalog allows specifying and checking higher-level abstract policy specifications that one can think of as operator beliefs [13]. Beliefs by and large hold, but may fail because of bugs or exceptions we wish to discover. For example, a common belief in our network is "Management stations should not be reachable from customer VMs or external Internet addresses". This is an instance of what we call a *Protection Set* template: "Stations in Set A cannot reach Stations in Set B".

For this paper, a belief is a Boolean combination of reachability predicates expressed using Datalog definitions. Rather than the operator enumerate the crossproduct of all specific reachability predicates between specific customer VM prefixes and all possible management stations, it takes less manual effort to state such beliefs. If the engine is armed with a mapping from the predicate "customer VM", "Internet", etc. to a list of prefixes, then the tool can unroll this more abstract policy specification into specific invariants between pairs of address prefixes. Of course, checks can lead to false positives (if there are exceptions which require refining our beliefs) or false negatives (if our list is incomplete), but we can start without waiting for perfect knowledge.

We have found five abstract policy templates (Table 1) cover every policy our operators have enforced and our security auditors check for: protection sets, reachability sets, reachability consistency, middlebox processing, and locality. An example of a reachability consistency belief and bug is shown in Figure 2. We will describe these in detail in the next section but Table 1 summarizes examples. Most are present in earlier work except for locality and reachability consistency. Despite this, earlier work in network verification with the exception of NetPlumber [22] does not allow reachability invariants to be specified at this level of abstraction. We make no attempt to learn these abstract policies as in [2, 13]. Instead, we glean these templates from talking to operators, encode them using NoD, and determine whether violations are exceptions or bugs by further discussion with operators.

A2. General modeling language to model networks: We provide Datalog as a tool not just to write the specification but also to write the router forwarding model. Thus it is easy for users to add support for MPLS or any new packet header such as VXLAN without changing internals. Second, we can model new forwarding behaviors enabled by programmable router languages such as P4 [4]. Third, we can model failure at several levels. The easiest level is not to model the routing protocol. For example, our links and devices are divided into availability zones that share components such as wires and power supplies that can fail together. Will an availability zone failure disconnect the network? This can be modeled by adding a predicate to each component that models its availability zone and state, and changing forwarding rules to drop if a component is unavailable.

Policy Template	Example	Datalog Feature Needed
Protection Sets	Customer VMs cannot access controllers	Definitions of sets
Reachable Sets	Customer VMs can access VMs	Definitions, Negation
Reachability Consistency	ECMP/Backup routes should have identical reachability/same path length	Negation, Non-determinism, Bit vectors
Middlebox processing	Forward path connections through a middlebox should reverse	Negation
Locality	Packets between two stations in the same cluster should stay within the cluster	Boolean combinations of reachability predicates

Table 1: 5 common policy templates.

At a second level, we can easily model failure response where the backup routes are predetermined as in MPLS fast reroute: when one tunnel in a set of equal cost tunnels fails, the traffic should be redistributed among the live tunnels in proportion to their weights. The next level of depth is to model the effect of route protocols like OSPF and BGP as has been done by Fogel et al. [15] for a university network using an early version of NoD. All of these failure scenarios can be modeled as Datalog rules, with routing protocol modeling [15] requiring that rules be run to a fixed point indicating routing protocol convergence. Our tool also can be used to ask analogous "what if" questions for reliability or security of network paths. By contrast, existing tools for network verification like VeriFlow and NetPlumber cannot model dynamism without changing internals. This is because the network model is hardcoded in these tools.

In summary, we need a verification engine that can specify beliefs and has the ability to model dynamism such as new packet headers or failures. Existing verification network tools scale to large networks at high speeds but with the exception of NetPlumber [22] do not have a specification language to specify beliefs. NetPlumber's regular expression language for reachability predicates, however, is less rich than Datalog; for example, it cannot model reachability consistency across ECMP routes as in Figure 2. More importantly, none of the existing tools including NetPlumber can model changes in the network model (such as failures) without modifying the internals.

On the other hand, the verification community has produced an arsenal of tools such as model checkers, Datalog, and SAT Solvers that are extremely extensible accompanied by general specification languages such as temporal logic. The catch is that they tend to work well only with small state spaces. For example, several Datalog implementations use relational backends that use tables as data structures. If the solutions are sets of headers and the headers can be 100 bits long, the standard implementations scale horribly. Even tools such as Margrave [27] while offering the ability to ask "what if" questions for firewalls do not scale to enumerating large header spaces. Thus, our contributions are:

1. Modeling Beliefs and Dynamism (§ **3**)**:** We show how to encode higher-level beliefs that can catch concrete bugs using succinct Datalog queries.

2. Network Optimized Datalog (§ **4**)**:** We modify the Z3 Datalog implementation by adding new optimizations such as symbolic representation of packets and a combined Filter-Project operator. While we will attempt to convey the high-level idea, the main point is that these optimizations are crucial to scale to large data centers with 100,000s of rules. Our code is publicly released as part of Z3 so that others can build network verification tools on top of our engine.

3. Evaluation (§ **6**): We show that the generality of Network Optimized Datalog comes at reasonable speeds using existing benchmarks, synthetic benchmarks, and our own operational networks. We also report briefly on the performance of other tools such as model checkers and SAT solvers.

4. Experience (§ **7**): We describe our experiences during the last year with an existing checker called SecGuru. We also describe our experience checking for beliefs on a large data center. Finally, we allude to Batfish [15] that checks for bugs in the face of failures using our tool as a substrate.

In addition, § 2 describes our Datalog model, and § 5 describes our benchmarks.

2 Datalog Model

2.1 Why Datalog

An ideal language/tool for network verification should possess five features:

1. All Solutions: We want to find all packet headers from *A* that can reach *B*. In other words, we need *all* solutions for a reachability query. Most classical verification tools such as model checkers [21] and SAT solvers [3] only provide single solutions; the naive approach of adding the negation of the solution and iterating is too slow.

2. Packet Rewrites: Among classical verification logics, Datalog does provide a native way to model routers as relations over input and output packets. Packet reachability is modeled as a recursive relation. Rewriting few selected bits or copying a range is also simple to model in this framework.

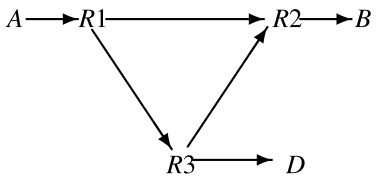
3. Large Header Spaces: Both Features 1 and 2 are challenging when the header space is very large; for example with packet headers of 80 bytes, the header space is of up to 2^{640} bits. Without a way to compress headers, naive solutions will scale poorly.

4. General Specification Language: Encoding beliefs minimally requires a language with Boolean operators for combining reachability sets, and negation to express differential queries.

5. General Modeling Language: Modeling failure response to protocols such as BGP and OSPF requires the ability to model recursion and running solutions to a fixed point as has been done by [15]. The language must also allow modeling simple failure scenarios such as availability zone failures and packet format changes.

Recent work in network verification including VeriFlow [24] and NetPlumber [22] provide all solutions, while allowing packet rewriting and scaling to large header spaces (Features 1-3). However, none of these domain-specific languages support Feature 5 (modeling any changes to network forwarding requires a change to tool internals) or Feature 4 (the specifications are typically hardcoded in the tool). While the FlowExp reachability language in NetPlumber [22] allows combination of reachability sets using regular expressions, it is fairly limited (e.g., it cannot do differential queries across paths) and is missing other higher-level features (e.g., recursion). By contrast, FlowExp queries can be encoded using Datalog, similar to the way finite automata are encoded in Datalog.

Thus, existing network verification languages [22–24, 26] fail to provide Features 4 and 5 while existing verification languages fail to provide Features 1, 2, and 3. FlowLog [28] is a language for programming controllers and seems less suited for verifying dataplanes. Out of the box Datalog is the only existing verification language that provides Features 1, 4, and 5. Natively, Datalog implementations struggle with Features 2 and 3. We deal with these challenges by overhauling the underlying Datalog engine (§ 4) to create a tool we call Network Optimized Datalog (NoD).



in	*dst*	*src*	*rewrite*	*out*
$R1$	$10\star$	$01\star$		$R2$
$R1$	$1\star\star$	$\star\star\star$		$R3$
$R2$	$10\star$	$\star\star\star$		B
$R3$	$\star\star\star$	$1\star\star$		D
$R3$	$1\star\star$	$\star\star\star$	$dst[1] := 0$	$R2$

Figure 1: $R1$ has a QoS routing rule that routes packets from a video source on the short route and other packets to destination $1\star\star$ along a longer path that traverses $R3$. $R3$ has an ACL that drops packets from $1\star\star$. $R3$ also rewrites the middle bit in *dst* to 0. This ensures that rerouted packets reach B regardless of the value of $dst[1]$.

2.2 Modeling Reachability in NoD

NoD is a Datalog implementation optimized for large header spaces. At the language level, NoD is just Datalog. We model reachability in NoD as follows.

Figure 1 shows a network with three routers $R1$, $R2$, and $R3$, and three end-point nodes A, B and D. The routing tables are shown below the picture. For this simple example, assume packets have only two fields *dst* and *src*, each a bit vector of 3 bits. When there are multiple rules in a router, the first matching rule applies. The last rule of Figure 1 includes a packet rewrite operation. We use $dst[1] := 0$ to indicate that position $dst[1]$ is set to 0. (Position 0 corresponds to the right-most bit.)

The goal is to compute the set of packets that can reach from A to B. For this example, the answer is easy to compute by hand and is the set of 6-bit vectors

$$10\star 01\star \cup (10\star\star\star\star \setminus \star\star\star 1\star\star)$$

where each packet is a 6-bit vector defined by a 3-bit value for *dst* followed by a 3-bit value for *src*, $\star$ denotes either 0 or 1, and $\setminus$ denotes set difference.

For reachability, we only model a single (symbolic) packet starting at the source. The current location of a packet is modeled by a location predicate. For example, the predicate $R1(dst,src)$ is true when a packet with destination *dst* and source *src* is at router $R1$.

Forwarding changes the location of a packet, and rewriting changes packet fields. A Datalog rule consists of two main parts separated by the :- symbol. The part to the left of this symbol is the `head`, while the part to the right is the `body` of the rule. A rule is read (and can be intuitively understood) as "`head` holds if it is known that `body` holds". The initial state/location of a packet is a *fact*, i.e., a rule without a `body`. For example, $A(dst,src)$ states that the packet starts at location A with destination address *dst* and source address *src*.

$$\begin{array}{rcl}
G_{12} & := & dst = 10\star \wedge src = 01\star \\
G_{13} & := & \neg G_{12} \ \wedge \ dst = 1\star\star \\
G_{2B} & := & dst = 10\star \\
G_{3D} & := & src = 1\star\star \\
G_{32} & := & \neg G_{3D} \ \wedge \ dst = 1\star\star \\
Id & := & src' = src \ \wedge \ dst' = dst \\
Set0 & := & src' = src \ \wedge \ dst' = dst[2]\ 0\ dst[0]
\end{array} \tag{1}$$

$$\begin{array}{rcl}
 & & B(dst,src) \\
R1(dst,src) & :- & G_{12} \wedge Id \wedge R2(dst',src') \\
R1(dst,src) & :- & G_{13} \wedge Id \wedge R3(dst',src') \\
R2(dst,src) & :- & G_{2B} \wedge Id \wedge B(dst',src') \\
R3(dst,src) & :- & G_{3D} \wedge Id \wedge D(dst',src') \\
R3(dst,src) & :- & G_{32} \wedge Set0 \wedge R2(dst',src') \\
A(dst,src) & :- & R1(dst,src) \\
 & ? & A(dst,src)
\end{array} \tag{2}$$

We use a shorthand for predicates that represent the matching condition in a router rule called a *guard* and for packet updates. The relevant guards and updates from Fig. 1 are in equation (1). Notice that G_{13} includes the negation of G_{12} to model the fact that the rule forwarding packets from $R1$ to $R3$ has lower priority than the one forwarding packets from $R1$ to $R2$. The update from the last rule (*Set0*) sets dst' to the concatenation of $dst[2]\ 0\ dst[0]$. Armed with this shorthand, the network of Fig. 1 can now be modeled as equation (2).

To find all the packets leaving A that could reach B, we pose the Datalog query $?A(dst,src)$ at the end of all the router rules. The symbol ? specifies that this is a query. Note how Datalog is used both as a modeling and a specification language.

Router FIBs and ACLs can be modeled by Datalog rules in a similar way. A router that can forward a packet to either $R1$ or $R2$ (load balancing) will have a separate (non-deterministic) rule for each possible next hop. We model *bounded* encapsulation using additional fields that are not used when the packet is decapsulated. Datalog queries can also check for cycles and forwarding loops. A loop detection query for an MPLS network is described below.

3 Beliefs and Dynamism in NoD

We now describe how to encode beliefs in NoD/Datalog, which either cannot be expressed succinctly, or at all, by previous work [23,24,26] without changing internals. While FlowExp in NetPlumber [22] allows modification of a reachability query *within a single path*, it cannot express queries *across paths* as Datalog can. We now show how to write Datalog queries for the belief templates alluded to in the introduction.

3.1 Protection sets

Consider the following belief: **Fabric managers are *not* reachable from guest virtual machines**. While this can be encoded in existing tools such as Hassel [23] and VeriFlow [24], since the guest VMs are set of size 5000, the fabric manager is a set of around 12 addresses, and the naive way to express this query is to explode the query to around $60,000$ separate queries. While this can be reduced by aggregating across routers, it is still likely to take many queries using existing tools [23, 24]. While this could be fixed by adding a way to define sets in say Hassel, this requires subtle changes to the internals. Our point is that Datalog allows this to be stated succinctly in a single query by taking advantage of the power of definitions.

The compact encoding in Datalog is as follows. Let $VM(dst,src)$ denote the fact that a packet is at one of the guest virtual machines and destined to an address dst that belongs to the set of fabric managers. We now query (based on the rules of the network, encoded for example as in equation (2)) for a *violation* of the belief: whether the fabric manager can be reached (encoded by the predicate FM) from a customer VM.

$$\begin{array}{lcl} VM(dst,src) & :- & AddrOfVM(src), AddrOfFM(dst). \\ & ? & FM(dst,src). \end{array}$$

Datalog Features: definitions for sets of addresses.

3.2 Reachability sets

Consider the following belief: **All Fabric managers are reachable from jump boxes (internal management devices)**.

As before, we check for the corresponding violation of the belief (a bug). Namely, we can query for addresses injected from jump boxes J, destined for fabric manager FM that nevertheless do not reach FM.

$$\begin{array}{lcl} J(dst,src) & :- & AddrOfJ(src), AddrOfFM(dst). \\ & ? & J(dst,src) \wedge \neg FM(dst,src). \end{array}$$

Datalog Features: Definitions, negation.

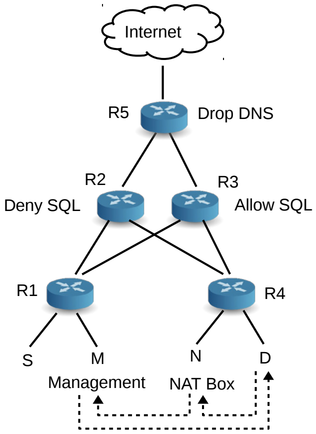


Figure 2: Bugs encountered in a cloud setting.

3.3 Equivalence of Load Balanced Paths

Consider the following bug:

Load Balancer ACL Bug: In Figure 2, an operator may set up two routers $R2$ and $R3$ that are load balancing traffic from a source S to destination D. $R2$ has an ACL entry that specifies that packets to the SQL port should be dropped but $R3$ does not. Assume that ECMP (equal cost multipath routing) [33] currently uses a hash function that routes SQL traffic via $R2$ where it is (correctly) dropped. However, the hash function can change to route packets via $R3$ and now SQL packets will (incorrectly) be let through.

In general, this bug violates a belief that *reachability across load balanced paths must be identical regardless of other variables such as hash functions*. We can check whether this belief is true by encoding a differential query encoded in Datalog. Is it possible that some packet reaches a destination under one hash function but not another? Such a query is impossible to answer using current network verification tools [22–24].

Datalog has the power needed for expressing reachability over different hashing schemes. We encode a hashing scheme as a bit vector h that determines what hashing choices (e.g., should $R1$ forward packets to D via $R2$ or $R3$ in Figure 2) are made at every router. We assume we have defined a built-in predicate *Select* that selectively enables a rule. We can then augment load balancing rules by adding *Select* as an extra guard in addition to the guards modeling the match predicate from the FIB. For example, if there are rules for routing from $R1$ to $R2$ and $R3$, guarded by G_{12}, G_{13}, then the modified rules take the form:

$$\begin{array}{lcl} R2(dst,h) & :- & G_{12} \wedge R1(dst,h) \wedge Select(h,dst). \\ R3(dst,h) & :- & G_{13} \wedge R1(dst,h) \wedge Select(h,dst). \end{array}$$

To check for inconsistent hashing, we pose a query that asks if there exists an intermediate node A at which packets to destination dst arrive using one hash assignment h_1 but are dropped using a different hash assignment h_2:

$$? \quad A(dst,h_1) \land \neg A(dst,h_2). \tag{3}$$

By adding an ordering across routers, the size of h can be encoded to grow linearly, not exponentially, with the path length.

Datalog Features: Negation, bit vectors, non-determinism, Boolean combinations of reachability predicates.

3.4 Locality

Consider the following bug:

Cluster reachability: In our Hong Kong data center, we found a locality violation. For example, it would be odd if packets from S to M in Figure 2 flowed through $R2$.

Putting aside the abstruse details of the wiring issues during buildout that caused this bug, the bug violates a belief that routing preserves traffic locality. For example, traffic within one rack must not leave the top-of-rack switch. Datalog definitions let us formulate such queries compactly.

Consider packets that flow between S and M in Figure 2. Observe that it would be odd if these packets flowed through $R2$, or $R3$, or $R5$ for that matter. The ability to define entire sub-scopes in Datalog comes in handy in this example. For example, we can define a predicate *DSP* (for *D*ata center *SP*ine) to summarize packets arriving at these routers:

$$\begin{aligned} DSP(dst) &\ :- \ R2(dst). \\ DSP(dst) &\ :- \ R3(dst). \\ DSP(dst) &\ :- \ R5(dst). \end{aligned}$$

Conversely, local addresses like S and M that can be reached via $R1$ can be summarized using another predicate L_{R1} (*L*ocal $R1$ addresses), that we assume is given as an explicit set of IP address ranges.

$$L_{R1}(dst) \ :- \ dst = 125.55.10.0/24.$$

Assume a packet originates at S and sends to such a local address; we ask if *DSP* is reached indicating that the packet has (incorrectly) reached the spine.

$$\begin{aligned} S(dst) &\ :- \ L_{R1}(dst). \\ &\ ? \ \ DSP(dst). \end{aligned} \tag{4}$$

Note the power of Datalog definitions. The query can also be abstracted further to check whether traffic between N and D (that sit on different racks and could be in a different cluster) have the same property in a single query, without writing separate queries for each cluster. For example, we could add rules, such as:

$$\begin{aligned} L_{R4}(dst) &\ :- \ dst = 125.75.10.0/24. \\ D(dst) &\ :- \ L_{R4}(dst). \\ N(dst) &\ :- \ L_{R4}(dst). \end{aligned}$$

This query can be abstracted further by defining locality sets and querying whether any two stations in the same locality set take a route outside their locality set.

Datalog Features: Scoping via predicates.

3.5 Dynamic Packet Headers

Going beyond encoding beliefs, we describe an example of dynamism. Network verification tools such as Hassel [23] and NetPlumber [22] support IP formats but do not yet support MPLS [31].

However, in Datalog one does not require *a priori* definitions of all needed protocols headers before starting an analysis. One can easily define new headers *post facto* as part of a query. More importantly, one can also *define new forwarding behaviors* as part of the query. This allows modeling flexible routers whose forwarding behavior can be metamorphosed at run-time [4, 5]

To illustrate this power, assume that the Datalog engine has no support for MPLS or the forwarding behavior of label stacking. A bounded stack can be encoded using indexed predicates. For example, if $R1$ is a router, then $R1^3$ encodes a forwarding state with a stack of 3 MPLS labels and $R1^0$ encodes a forwarding state without any labels. Using one predicate per control state we can encode a forwarding rule from $R5$ to $R2$ that pushes the label 2016 on the stack when the guard G holds as:

$$\begin{aligned} R2^1(dst,src,2016) &\ :- \ G, R5^0(dst,src). \\ R2^2(dst,src,l_1,2016) &\ :- \ G, R5^1(dst,src,l_1). \\ R2^3(dst,src,l_1,l_2,2016) &\ :- \ G, R5^2(dst,src,l_1,l_2). \\ Ovfl(dst,src,l_1,l_2,l_3) &\ :- \ G, R5^3(dst,src,l_1,l_2,l_3). \end{aligned}$$

We assume that l_1, l_2, l_3 are eight bit vectors that model MPLS labels. The first three rules model both MPLS label stacking procedure and format. Predicates, such as $R1^3$, model the stack size. The last rule checks for a misconfiguration that causes label stack overflow.

SWAN [19] uses an MPLS network updated dynamically by an SDN controller. To check for the belief that the SDN controller does not create any loops, we use standard methods [37]. We use a field to encode a partial history of a previously visited router. For every routing rule, we create two copies. The first copy of the rule sets the history variable h to the name of the router. The other copy of the rule just forwards the history variable. There is a loop (from $R5$ to $R5$) if $R5$ is visited again where the history variable holds $R5$. We omit details.

While router tables [19] may be in flux during updates, a reasonable belief is even during updates "rules do not overlap", to avoid forwarding conflicts. Similarly, another reasonable belief is that any rule that sends a packets out of the MPLS network should pop the *last* element from the MPLS label stack.

3.6 Middleboxes and Backup Routers

While ensuring that traffic goes through a middlebox M is a well-studied property [22], we found a more subtle bug across paths:

Incorrect Middlebox traversal: Once again, refer to Figure 2. A management box M in a newer data center in Brazil attempted to start a TCP connection to a local host D. Newer data centers use a private address space and internal hosts must go through a Network Address Translator (NAT) before going to external public services. The box M sent the TCP connection request to the private address of D, but D sends the packet to M via the NAT which translates D's source address. TCP at M reset the connection because the SYN-ACK arrived with an unexpected source.

This bug violates a belief that packets should go through the same set of middleboxes in the forward and reverse path. This is a second example of an invariant that relates reachability across two different paths, in this case the forward and reverse paths, as was reachability across load balanced paths. This is easily encoded in Datalog by adding a fictitious bit to packets that is set when the packet passes through a middlebox. We omit details to save space.

Consider next the following bug:

Backup Non-equivalence: Two data centers $D1$ and $D2$ in the same region are directly connected through a border network by a pair of backup routers at the border of $D1$ and $D2$. The border routers are also connected to the core network. The intent is that if a single failure occurs $D1$ and $D2$ remain directly connected. However, we found after a single failure, because of an incorrect BGP configuration, the route from $D1$ to $D2$ went through a longer path through the core. While this is an inefficiency bug, if it is not fixed and then if the route to the core subsequently fails, then $D1$ and $D2$ lose connectivity even when there is a physical path.

This bug suggests the belief that all paths between a source and destination pair passing through any one of a set of backup routers should have the same number of hops. We omit the encoding except to note that we encode path lengths in Datalog as a small set of control bits in a packet, and query whether a destination is reached from the same source across one of the set of backup routers, but using two different path lengths. Once again this query appears impossible to state with existing network verification tools except NetPlumber [22]. NetPlumber, however, cannot handle the range of queries NoD can, especially allowing dynamic networks.

4 Network Optimized Datalog

While Datalog can express higher-level beliefs and model dynamism (Features 4 and 5 in Section 2.1) and computes all solutions (Feature 1), naive Datalog implementations struggle with Features 2 and 3 (scalably expressing large header spaces and packet rewrites). While we describe our experience with modifying μZ (the Datalog framework in Z3), there are two general lessons that may apply to other Datalog tools: the need for a new table data structure to compactly encode large header spaces, and a new Select-Project operator. We will try to convey the high-level ideas for a networking reader.

4.1 Compact Data Structures

One can pose reachability queries to μZ (our Datalog framework) to compute the set of packets that flow from A to B. Think of the Datalog network model as expressing relations between input and output packets at each router: the router relation models the forwarding behavior of the router including all forwarding rules and ACLs. The router is modeled not as a function but as a relation, to allow multicast and load balancing, where several output packets can be produced for the same input packet. Whenever the set of input packets at a router change, the corresponding set of output packets are recomputed using the router relation. Thus for a network, eventually sets of packets will flow from the inputs of the network to the endpoints of the network.

The main abstract data structure to encode a relation in Datalog is a table. For example, a table is used to store the set of input packets at a router, and a table is used to store output packets. To update the relationship between input and output tables at a router, under the covers, μZ executes Datalog queries by converting them into relational algebra, as described elsewhere [9]. Networking readers can think of μZ as providing a standard suite of database operators such as *select*, *project* and *join* to manipulate *tables* representing sets of packet headers in order to compute reachability sets. Figure 3 – which represents a single router that drops HTTP packets using an ACL that drops port 80 packets – makes this clearer.

In our toy example, Figure 3, assume that the set of packets that reach this router have source addresses whose first bit is 1 because of some earlier ACLs. Thus the set of packet headers that leave the router are those that have first bit 1 and whose TCP destination port is

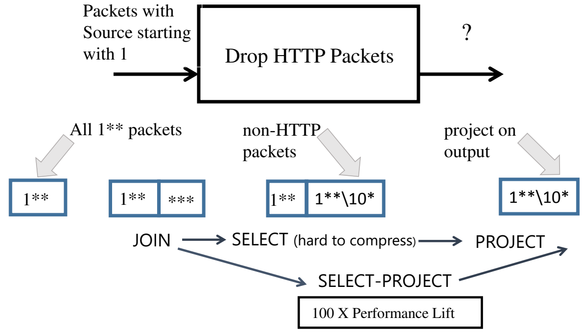


Figure 3: Manipulating tables of packet headers with a combined select-project operator.

not 80. As we have seen, the natural way for Datalog to represent a set of packet headers is as a *table*. Representing all source addresses that start with a 1 would require 2^{127} rows if 128-bit packet headers are represented by arrays. None of the existing table data structures (encapsulated in what are called backends) in μZ performed well for this reason. Hence, we implemented two new table backends.

The first backend uses BDDs (Binary Decision Diagrams [7]) for Datalog tables. BDDs are a classic data structure to compactly represent a Boolean function, and widely used in hardware verification to represent circuits [8]. A classic program analysis paper also augments Datalog with BDDs [34] for program analysis, so our use of BDDs for Datalog tables is natural.

The second backend is based on ternary bit vectors, inspired by Header Space Analysis (HSA) [23, 24], but placed in a much more general setting by adding a new data structure to Datalog. This data structure we added was what we call *difference of cubes* or DoC. DoC represents sets of packets as a *difference* of ternary strings. For example, $1\star\star \setminus 10\star$ succinctly represents all packets that start with 1 other than packets that start with 10. Clearly, the output of Figure 3 can be compactly represented as the set difference of all $1\star\star$ packets and all packets whose destination port is 80.

More precisely, for ternary bit vectors v_i and v_j, a difference of cubes represents a set

$$\bigcup_i \left(v_i \setminus \bigcup_j v_j \right)$$

The difference of cubes representation is particularly efficient at representing router rules that have dependencies. For example, the second rule in Figure 1 takes effect only if the first rule does not match. More precisely, difference of cubes is particularly efficient at representing formulas of the form $\varphi \wedge \neg\varphi_1 \wedge \cdots \wedge \neg\varphi_n$, with φ and φ_i of the form $\bigwedge_i \phi_i$ and ϕ_i having no Boolean operators. This form is precisely what we obtain in the transfer functions of routing rules, with φ being the route matching formula, and the $\neg\varphi_i$ being the negation of the matching formula of the dependencies of the rule. Again, in networking terminology dependencies of a forwarding rule or ACL are all higher priority matching rules.

Code: The Datalog backends added to Z3 were implemented in C++. The BDD backend takes 1,300 LoC, and the difference of cubes backend takes almost 2,000 LoC.

4.2 Combining Select and Project

We needed to go beyond table compression in order to speedup Datalog's computation of reachability sets as follows. Returning to Figure 3, μZ computes the set of output packets by finding a relation between input packets and corresponding output packets. The relation is computed in two steps: first, μZ *joins* the set of input packets I to the set of all possible output packets A to create a relation (I, A). Next, it *selects* the output packets (rows) that meet the matching and rewrite conditions to create a pruned relation (I, O). Finally, it *projects* away input packets and produces the set of output packets O.

Thus, in Figure 3, the output of the first *join* is the set of all possible input packets with source addresses that start with 1, together with all possible output packets. While this sounds like a very indirect and inefficient path to the goal, this is the natural procedure in μZ. Joins are the only way to create a new relation, and to avail of the powerful set of operators that work on relations. While the join with all possible output packets A appears expensive, A is compactly represented as a single ternary string/cube and so its cost is small.

Next, after the *select*, the relation is "trimmed" to be all possible input packets with source bit equal to 1, together with all output packets with source bit equal to 1 and destination port not equal to 80. Finally, in the last step, the *project* step removes all columns corresponding to the input packets, resulting in the correct set of output packets (Figure 3) as desired. Observe that the output of the *join* is easily compressible (a single ternary string) and the output of the final *project* is also compressible (difference of two ternary strings).

The elephant in the room is the output of the *select* which is extremely inefficient to represent as a BDD or as a difference of ternary strings. But the output of the *select* is merely a way station on the path to the output; so we do not need to explicitly materialize this intermediate result. Thus, we define a new combined *select-project* operator whose inputs and outputs are both compressible. This is the key insight, but making it work in the presence of packet rewriting requires more intricacy.

4.3 Handling Packet Rewriting

The example in Figure 3 does not include any packet rewrites. In Figure 1, however, rule R_3 rewrites the first destination address bit to a 0. Intuitively, all bits *except* the first are to be copied from the input packet to the output. While this can be done by a brute force enumeration of all allowed bit vectors for this 6-bit toy example, such an approach does not scale to 128 bit headers. We need, instead, an efficient way to represent copying constraints.

Back to the toy example, consider an input packet $1\star\star\star\star\star$ at router R_3 that is forwarded to router R_2. Recall that the first 3 bits in this toy example are the destination address, the next 3 are the source address. We first join the table representing input packets with a full table (all possible output packets), obtaining a table with the row $1\star\star\star\star\star\star\star\star\star\star\star$, where the first six bits correspond to the input packet at R_3, and the remaining six bits belong to the output destined to R_2.

Then we apply the guard and the rewrite formulas and the negation of all of the rule's dependencies using a generalized select). Since we know that the 5th rule in Figure 1 can only apply if 4th rule does not match, we know that in addition to the first destination address bit being 1, the "negation of the dependencies" requires that bit 2 of the source address should also be 0.

One might be tempted to believe that $1\star\star\star\star\star10\star0\star\star$ compactly represents the input-output relation as "All input packets whose second destination address bit is 1" (the first six bits) together with "All output packets for which bit 2 of destination address bit is 1, bit 1 of the destination address has been set to 0, and for which bit 2 of the source address bit is 0". But this incorrectly represents the copying relation! For example, it allows input packets where bit 1 of the source address of the input packet is a 0, but bit 1 of the source address of the output packet is a 1. Fortunately, we can rule out these exceptions fairly compactly using set differences which are allowed in difference of cubes notation. We obtain the following expression:

$$\begin{array}{l}
1\star\star\star\star\star10\star0\star\star \setminus \\
(\star\star0\star\star\star\star\star1\star\star\star\cup\star\star1\star\star\star\star\star0\star\star\star\cup \\
\;\star\star\star\star0\star\star\star\star\star1\star\cup\star\star\star\star1\star\star\star\star\star0\star\cup \\
\;\star\star\star\star\star0\star\star\star\star\star1\cup\star\star\star\star\star1\star\star\star\star\star0)
\end{array}$$

While this looks complicated, *the key idea is efficiently representing copying using a difference of ternary strings*. The unions in the difference are ruling out cases where the "don't care" $\star$ bits are not copied correctly. The first term in the difference states that we cannot have bit 0 of destination address bit be a 0 in the *input* packet and bit 0 of destination address bit in the *output* packet be a 1; the next term disallows the bits being 1 and 0 respectively. And so on for all the bit positions in *dst* and *src* whose bits are being copied.

After the select operation, we perform a projection to remove the columns corresponding to the input packet (the first 6 bits) and therefore obtain a table with only the output packets. Again, in difference of cubes representation, we obtain $10\star0\star\star$. The final result is significantly smaller than the intermediate result, and this effect is much more pronounced when we use 128 bit headers!

Generalizing: To make select-project efficient, we need to compute the projection implicitly without explicitly materializing intermediate results. We did this using a standard union-find data structure to represent equivalence classes (copying) between columns. When establishing the equality of two columns, if both columns (say bit 3 of the Destination address in both input and output packets) contain "don't care" values and one of them (bit 3 in the input packet) will be projected out, we aggregate the two columns in the same equivalence class. While this suffices for networking, we added two more rules to generalize this construction soundly to other domains besides networking. In verification terminology, this operation corresponds to computing the strongest post-condition of the transition relation. However, it takes some delicacy to implement such an operator in a general verification engine such as Z3 so it can be used in other domains besides network verification.

Code: Besides select-project, we made several additional improvements to the Datalog solver itself, which were released with Z3 4.3, reducing Z3's memory usage in our benchmarks by up to 40% .

5 Benchmarks

We use four benchmarks:

Stanford: This is a publicly available [32] snapshot of the routing tables of the Stanford backbone, and a set of network reachability and loop detection queries. The core has 16 routers. The total number of rules across all routers is 12,978, and includes extensive NAT and VLAN support.

Generic Cloud Provider: We use a parameterizable model of a cloud provider network with multiple data centers, as used by say Azure or Bing. We use a fat tree as the backbone topology *within* a data center and single top-of-rack routers as the leaves. Data centers are interconnected by an all-to-all one-hop mesh network. Parameters include replication factor, router ports, data centers, machines per data center, VMs per machine, and number of services. These parameters can be set to instantiate small, medium, or large clouds. This benchmark is publicly available [29].

Production Cloud: Our second source comes from two newer live data centers located in Hong Kong and in Singapore whose topology is shown in Figure 4. They

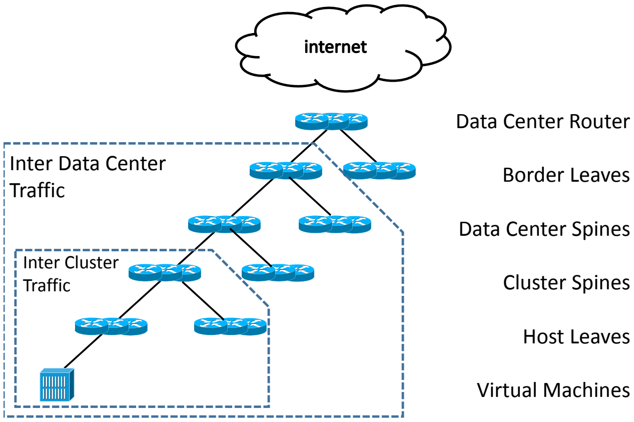


Figure 4: Production Cloud Layout

consist of a hundred routers each ranging over border leaves, data center and cluster spines, and finally top-of-rack switches. In the Hong Kong data center, each router has roughly 2000 ECMP forwarding rules adding up to a combined 200K rules. In the Singapore data center there are about 820K combined forwarding rules. The text files for these rules take from 25MB to 120MB for the respective data-centers.

This infrastructure services three clusters and thousands of machines. We extracted routing tables from the Arista and Cisco devices using a `show ip route` command. This produced a set of routing tables including the ECMP routing options in Datalog format. To handle longest prefix match semantics, the translation into Datalog uses a trie to cluster ranges with common prefixes, avoiding redundancy in the Datalog rules. The range of private and public IP addresses assigned to each cluster was extracted from a separate management device.

Experimental Backbone: We check an SDN backbone based on the SWAN design [19]. To maximize bandwidth utilization, the SDN controller periodically recomputes routing rules that encapsulate packets with an MPLS labels stack encoding tunnels. Our tool takes the output from the controller: a set of routing tables, a network topology and configurations that map IP addresses to end-points. We check selected beliefs, such as loop freedom and absence of stack overflow.

Experimental Toolkit: In addition to the publicly available Hassel C code [16], for comparison we used two classic model checking algorithms (BMC and PDR), a Datalog framework μZ [18], and a state-of-the-art SAT/SMT solver, all implemented in the Z3 [11] engine. Our engines are publicly available [29] so other researchers can extend our results.

6 Evaluation

We describe results for belief checking on a production cloud IP network and an experimental MPLS backbone, followed by differential reachability queries on a cloud benchmark. Finally, we compare NoD's performance with existing tools.

6.1 Protection Sets in a Production Cloud

We checked whether two policies based on the Protection sets template (see Table 1) hold in the Singapore data center. The two queries were to verify that neither Internet addresses or customer VMs can access the protected fabric controllers for security reasons.

Both experiments took around 12 minutes. While this may seem slow, the sets of addresses are very large. For the power of exploring a very general belief, the performance seems acceptable and easily incorporated in a checker that runs every hour.

Both queries failed; thus the beliefs of the network operators were incorrect. Closer inspection showed that these were not network bugs, but incorrect beliefs. There are two ranges of IPs of fabric controllers that are supposed to be reachable (and they are), and certain ICMP packets are also allowed to flow to fabric controllers. Although no network bugs were found, these queries allowed the operator to refine his beliefs.

6.2 Reachable Sets on a Production Cloud

As in the previous experiment, we also checked whether two policies from the reachable sets template hold in the same Singapore data center. The first query checked if all of "utility boxes" can reach all "fabric controllers". The second query is similar, but checks whether "service boxes" can reach "fabric controllers". The first query took around 4 minutes to execute, while the second took 6 minutes. Both queries passed successfully, confirming operator beliefs.

6.3 Locality on a Production Cloud

Figure 4 shows our public cloud topology which has more levels of hierarchy than the simple example of Figure 2. These levels motivate more general queries than the simple locality query (4) used in [§3.4].

Figure 4 also shows the boundaries of traffic locality in our public cloud. Based on these boundaries, we formulate the following queries to check for traffic locality. First, C2C requires that traffic within a cluster not reach beyond the designated cluster spines. Next, B2DSP, B$\bar{2}$DSP, B2CSP, and B$\bar{2}$CSP require that traffic targeting public addresses in a cluster must reach only designated data center spines, not reach other data center spines belonging to a different DC, must reach only designated cluster spines, and not reach cluster spines belonging to other clusters, respectively.

Query	Cluster 1	Cluster 2	Cluster 3
C2C	12 (2)	13 (2)	11 (2)
B2DSP	11 (2)	11 (2)	11 (2)
B$\bar{2}$DSP	3 (1)	4 (1)	4 (1)
B2CSP	11 (2)	11 (2)	11 (2)
B$\bar{2}$CSP	11 (2)	12 (2)	11 (2)

Table 2: Query times are in seconds, times in parentheses are for a snapshot where only one of the available ECMP options is part of the model (20% the size of the full benchmarks).

Table 2 shows the time spent for these five different kinds of queries over three clusters. They check correctness of routing configurations in all network devices.

For the Hong Kong cluster we identified some violated locality queries due to bugs during buildout. An example output for a C2C query was 100.79.126.0/23 suggesting that an address range got routed outside the cluster it was said to be part of. On the other hand, our tool produced public address ranges from a B2CSP query which were supposed to reach the cluster spine but did not: $192.114.2.62/32 \cup \ldots \cup 192.114.3.48/29$.

6.4 An Experimental MPLS backbone

We checked loop-freedom, absence of black holes, and rule disjointness on configurations for an experimental backbone based on the SWAN design [19]. Note the versatility of Datalog and its ability to model dynamic forwarding behaviors without changing tool internals. Both packet formats and forwarding behaviors were completely different from those in the production cloud queries. We report on a selected experiment with a configuration comprising of 80 routers, for a total of 448 forwarding rules. Modelling label stacks required 3400 Datalog rules over a header-space of 153 bits. The loop check takes a fraction of a second for the configurations we checked. Checking for black holes takes under 5 seconds, identifying 56 flows out of 448 as black holes. The same configuration had 96 pairs of overlapping rules, which were enumerated in less than 1 second. This experience bodes well for scaling validation to larger backbone sizes.

6.5 Differential Reachability

We investigate the performance of differential reachability queries on our synthetic cloud benchmarks (topology similar to Figure 2).

We performed two experiments by taking the Medium Cloud topology as baseline, and changing the ACLs at one of the core routers such that one of the links in a set of load balanced paths allowed VLAN 3 and blocked VLAN 1, while all other links blocked VLAN 1 and allowed VLAN 3. We then checked the difference in reachability across all load-balanced paths between the Internet and a host in the data center.

This query took 1.9 s, while the equivalent reachability query takes 1.0 s (90% run time increase). Repeating the same query, but measuring differential reachability between two hosts located in different data centers (resulting in longer paths) took 3.1 s, while the equivalent reachability queries takes 1.1 s (182% run time increase).

We note that the run time increase between vanilla reachability queries and differential queries will likely increase with the amount of differences, but in practice the number of differences (i.e., bugs) should be small.

6.6 Comparison with existing Tools

We ran the benchmarks with multiple tools to benchmark our speeds against existing work. Besides the difference of cubes backend described in § 4, we also used a BDD-based Datalog backend, a bounded model checker (BMC) [10], an SMT solver that uses an unrolled representation of the network as in [37], and a state-of-the-art solver based on the IC3 [6, 17] algorithm. The benchmarks were run on a machine with an Intel Xeon E5620 (2.4 GHz) CPU. We also used an SMT algorithm that was modified to return all solutions efficiently. The model checkers, however, return only one solution so the speed comparison is not as meaningful.

Table 3 is a small sampling of extensive test results in [29]. It shows time (in seconds) to run multiple tools on a subset of the Stanford benchmarks, including reachable and unreachable queries, and a loop detection query. The tests were given a timeout of 5 minutes and a memory limit of 8 GBs.

Note that the model checkers only compute satisfiability answers (i.e., "is a node reachable or not?"), while Datalog computes all reachable packets. For SMT, we provide results for both type of queries. All the SMT experiments were run with the minimum TTL (i.e., an unrolling) for each test; for example, the TTL for Stanford was 3 for reachability and 4 for loop detection. Higher TTLs significantly increase running time. We do not provide the SMT running times for the cloud benchmarks, since our AllSAT algorithm does not support non-determinism. We were unable to run Hassel C [16] on the cloud benchmarks, since Hassel C has hardwired assumptions, such as router port numbers following a specific naming policy.

The first takeaway is that NoD is faster at computing *all solutions* than model checkers or SAT solvers are at computing *a single solution*. Model checking performance also seems to degrade exponentially with path length (see row 3 versus row 2 where the model checkers run out of memory). Similarly, unrolling seems to exact

Test	Model Checkers		SMT		Datalog		Hassel C
	BMC	PDR	Reach.	All sols.	BDDs	DoC	
Small Cloud	0.3	0.3	0.1	–	0.2	0.2	–
Medium Cloud	T/O	10.0	0.2	–	1.8	1.7	–
Medium Cloud Long	M/O	M/O	4.8	–	7.4	7.2	–
Cloud More Services	7.2	8.5	12.5	–	5.3	4.8	–
Large Cloud	T/O	M/O	2.8	–	16.1	15.7	–
Large Cloud Unreach.	T/O	M/O	1.1	n/a	16.1	15.7	–
Stanford	56.2	13.7	11.5	1,121	6.6	5.9	0.9
Stanford Unreach.	T/O	12.2	0.1	n/a	2.6	2.1	0.1
Stanford Loop	20.4	11.7	11.2	290.2	6.1	3.9	0.2

Table 3: Time (in seconds) taken by multiple tools to solve network benchmarks. Model checkers only check for satisfiability, while Datalog produces reachability sets. T/O and M/O are used for time- and memory-out.

a price for SMT solvers. Even our efficient AllSAT generalization algorithm is around 200× slower than Datalog (row 7). Datalog with difference of cubes is the most competitive implementation we have tested. Datalog with a BDD backend shows good performance as well.

Hassel C takes under a second for Stanford, faster than Datalog. NetPlumber [22] and VeriFlow [24] are even faster for incremental analysis. Further, Yang and Lam [36] use predicate abstraction to further speed up reachability testing. However, none of these tools have the ability to model higher-level beliefs and model dynamic networks as NoD can. NoD speeds are also acceptable for an offline network checker, the major need today.

7 Experience

SecGuru: The SecGuru [20] tool has been actively used in our production cloud. In continuous validation, SecGuru checks policies over ACLs on every router update as well as once a day, doing over 40,000 checks per month, where each check takes 150-600 ms. It uses a database of predefined common beliefs. For example, there is a policy that says that SSH ports on fabric devices should not be open to guest VMs. While these policies rarely change, IP addresses do change frequently, which makes SecGuru useful as a regression test. SecGuru had a measurable impact in reducing misconfigurations during build-out, raising an average of one alert per day, each identifying ∼16K faulty addresses. SecGuru also helped reduce our legacy corporate ACL from roughly 3000 rules to 1000 without any business impact.

Belief Refinement: As described in § 6.1, our operator's belief in a protection set policy (customer VMs cannot reach fabric controllers) was subtly incorrect. The correct belief required new reachability exceptions. We currently code this at the level of Datalog in lieu of a GUI interface to the 5 policy templates from Table 1.

Batfish: Batfish [15] can find whether two stations remain reachable across any set of single link failures by modeling OSI and BGP. Batfish uses NoD for reachability analysis because it is more expressive than other tools. Batfish also uses the Z3 constraint solver to find concrete packet headers, confirming our intuition that supplying NoD in a general tool setting allows unexpected uses.

8 Conclusion

Network Optimized Datalog (NoD) is more expressive than existing tools in its ability to encode beliefs to uncover true specifications, and to model network dynamism without modifying internals. NoD is much faster than existing (but equally expressive) verification tools such as model checkers and SMT solvers. Key to its efficiency are ternary encoding of tables and a Select-Project operator.

By contrast, current network verification tools operate at a low level of abstraction. Properties such as cluster scoping (Figure 4) expressible in a single Datalog query would require iterating across all source-destination pairs. Further, existing work cannot easily model new packet formats, new forwarding behaviors, or failures [15]. We were pleased to find we could model MPLS and label stacking succinctly. It took a few hours to write a query to find loops in SWAN.

As in [13], our work shows how fragile the understanding of the true network specification is. Even when working with an experienced network operator, we found that simple beliefs (e.g., no Internet addresses can reach internal controllers) had subtle exceptions. The belief templates in Table 1 abstract a vast majority of specific checks in our network and probably other networks. A GUI interface for belief templates will help operators.

If network verification is to mature into a networking CAD industry, its tools must evolve from ad hoc software into principled and extensible techniques, built upon common foundations that are constantly being improved. We suggest NoD as a candidate for such a foundation.

References

[1] E. Al-Shaer and S. Al-Haj. FlowChecker: configuration analysis and verification of federated OpenFlow infrastructures. In *SafeConfig*, 2010.

[2] T. Benson, A. Akella, and D. A. Maltz. Mining policies from enterprise network configuration. In *IMC*, 2009.

[3] A. Biere, A. Biere, M. Heule, H. van Maaren, and T. Walsh. *Handbook of Satisfiability*. IOS Press, 2009.

[4] P. Bosshart, D. Daly, G. Gibb, M. Izzard, N. McKeown, J. Rexford, C. Schlesinger, D. Talayco, A. Vahdat, G. Varghese, and D. Walker. P4: Programming protocol-independent packet processors. *SIGCOMM Comput. Commun. Rev.*, 44(3):87–95, July 2014.

[5] P. Bosshart, G. Gibb, H.-S. Kim, G. Varghese, N. McKeown, M. Izzard, F. Mujica, and M. Horowitz. Forwarding metamorphosis: Fast programmable match-action processing in hardware for SDN. In *SIGCOMM*, 2013.

[6] A. R. Bradley. SAT-based model checking without unrolling. In *VMCAI*, 2011.

[7] R. E. Bryant. Graph-based algorithms for boolean function manipulation. *IEEE Trans. Comput.*, 35(8):677–691, Aug. 1986.

[8] J. R. Burch, E. M. Clarke, K. L. McMillan, D. L. Dill, and L. J. Hwang. Symbolic model checking: 10^{20} states and beyond. In *LICS*, 1990.

[9] S. Ceri, G. Gottlob, and L. Tanca. What you always wanted to know about Datalog (and never dared to ask). *IEEE Trans. on Knowl. and Data Eng.*, 1(1):146–166, Mar. 1989.

[10] E. Clarke, A. Biere, R. Raimi, and Y. Zhu. Bounded model checking using satisfiability solving. *Form. Methods Syst. Des.*, 19(1):7–34, 2001.

[11] L. de Moura and N. Bjørner. Z3: an efficient SMT solver. In *TACAS*, 2008.

[12] D. Engler, B. Chelf, A. Chou, and S. Hallem. Checking system rules using system-specific, programmer-written compiler extensions. In *OSDI*, 2000.

[13] D. R. Engler, D. Y. Chen, and A. Chou. Bugs as deviant behavior: A general approach to inferring errors in systems code. In *SOSP*, 2001.

[14] C. Flanagan, K. R. M. Leino, M. Lillibridge, G. Nelson, J. B. Saxe, and R. Stata. Extended static checking for java. In *PLDI*, 2002.

[15] A. Fogel, S. Fung, L. Pedrosa, M. Walraed-Sullivan, R. Govindan, R. Mahajan, and T. Millstein. A general approach to network configuration analysis. In *NSDI*, 2015.

[16] Hassel C. `https://bitbucket.org/peymank/hassel-public/`.

[17] K. Hoder and N. Bjørner. Generalized property directed reachability. In *SAT*, 2012.

[18] K. Hoder, N. Bjørner, and L. De Moura. μZ: an efficient engine for fixed points with constraints. In *CAV*, 2011.

[19] C.-Y. Hong, S. Kandula, R. Mahajan, M. Zhang, V. Gill, M. Nanduri, and R. Wattenhofer. Achieving high utilization with software-driven WAN. In *SIGCOMM*, 2013.

[20] K. Jayaraman, N. Bjørner, G. Outhred, and C. Kaufman. Automated Analysis and Debugging of Network Connectivity Policies. Technical Report MSR-TR-2014-102, Microsoft Research, July 2014.

[21] R. Jhala and R. Majumdar. Software model checking. *ACM Comput. Surv.*, 41(4):21:1–21:54, Oct. 2009.

[22] P. Kazemian, M. Chang, H. Zeng, G. Varghese, N. McKeown, and S. Whyte. Real time network policy checking using header space analysis. In *NSDI*, 2013.

[23] P. Kazemian, G. Varghese, and N. McKeown. Header space analysis: static checking for networks. In *NSDI*, 2012.

[24] A. Khurshid, X. Zou, W. Zhou, M. Caesar, and P. B. Godfrey. VeriFlow: verifying network-wide invariants in real time. In *NSDI*, 2013.

[25] M. Mahalingam, D. Dutt, K. Duda, P. Agarwal, L. Kreeger, T. Sridhar, M. Bursell, and C. Wright. Virtual extensible local area network (VXLAN): A framework for overlaying virtualized layer 2 networks over layer 3 networks. RFC 7348, Aug. 2014.

[26] H. Mai, A. Khurshid, R. Agarwal, M. Caesar, P. B. Godfrey, and S. T. King. Debugging the data plane with Anteater. In *SIGCOMM*, 2011.

[27] T. Nelson, C. Barratt, D. J. Dougherty, K. Fisler, and S. Krishnamurthi. The Margrave tool for firewall analysis. In *LISA*, 2010.

[28] T. Nelson, A. D. Ferguson, M. J. G. Scheer, and S. Krishnamurthi. Tierless programming and reasoning for software-defined networks. In *NSDI*, 2014.

[29] Network verification website. `http://web.ist.utl.pt/nuno.lopes/netverif/`.

[30] P. Patel, D. Bansal, L. Yuan, A. Murthy, A. Greenberg, D. A. Maltz, R. Kern, H. Kumar, M. Zikos, H. Wu, C. Kim, and N. Karri. Ananta: Cloud scale load balancing. In *SIGCOMM*, 2013.

[31] E. Rosen, A. Viswanathan, and R. Callon. Multiprotocol label switching architecture. RFC 3031, Jan. 2001.

[32] Stanford benchmark. `https://bitbucket.org/peymank/hassel-public/src/697b35c9f17ec74ceae05fa7e9e7937f1cf36878/hassel-c/tfs/`.

[33] D. Thaler and C. Hopps. Multipath issues in unicast and multicast next-hop selection. RFC 2991, Nov. 2000.

[34] J. Whaley and M. S. Lam. Cloning-based context-sensitive pointer alias analysis using binary decision diagrams. In *PLDI*, 2004.

[35] G. G. Xie, J. Zhan, D. A. Maltz, H. Zhang, A. G. Greenberg, G. Hjálmtýsson, and J. Rexford. On static reachability analysis of IP networks. In *INFOCOM*, 2005.

[36] H. Yang and S. Lam. Real-time verification of network properties using atomic predicates. In *ICNP*, 2013.

[37] S. Zhang, S. Malik, and R. McGeer. Verification of computer switching networks: An overview. In *ATVA*, 2012.

C3: Cutting Tail Latency in Cloud Data Stores via Adaptive Replica Selection

Lalith Suresh[†] Marco Canini[⋆] Stefan Schmid[†‡] Anja Feldmann[†]
[†]*TU Berlin* [⋆]*Université catholique de Louvain* [‡]*Telekom Innovation Labs*

Abstract

Achieving predictable performance is critical for many distributed applications, yet difficult to achieve due to many factors that skew the tail of the latency distribution even in well-provisioned systems. In this paper, we present the fundamental challenges involved in designing a replica selection scheme that is robust in the face of performance fluctuations across servers. We illustrate these challenges through performance evaluations of the Cassandra distributed database on Amazon EC2. We then present the design and implementation of an adaptive replica selection mechanism, C3, that is robust to performance variability in the environment. We demonstrate C3's effectiveness in reducing the latency tail and improving throughput through extensive evaluations on Amazon EC2 and through simulations. Our results show that C3 significantly improves the latencies along the mean, median, and tail (up to 3 times improvement at the 99.9^{th} percentile) and provides higher system throughput.

1 Introduction

The interactive nature of modern web applications necessitates low and predictable latencies because people naturally prefer fluid response times [20], whereas degraded user experience directly impacts revenue [11, 43]. However, it is challenging to deliver consistent low latency — in particular, to keep the tail of the latency distribution low [16, 23, 48]. Since interactive web applications are typically structured as multi-tiered, large-scale distributed systems, even serving a single end-user request (e.g., to return a web page) may involve contacting tens or hundreds of servers [17, 23]. Significant delays at any of these servers inflate the latency observed by end users. Furthermore, even temporary latency spikes from individual nodes may ultimately dominate end-to-end latencies [2]. Finally, the increasing adoption of commercial clouds to deliver applications further exacerbates the response time unpredictability since, in these environments, applications almost unavoidably experience performance interference due to contention for shared resources (like CPU, memory, and I/O) [26, 50, 52].

Several studies [16, 23, 50] indicate that latency distributions in Internet-scale systems exhibit long-tail behaviors. That is, the 99.9^{th} percentile latency can be more than an order of magnitude higher than the median latency. Recent efforts [2, 16, 19, 23, 36, 44, 53] have thus proposed approaches to reduce tail latencies and lower the impact of skewed performance. These approaches rely on standard techniques including giving preferential resource allocations or guarantees, reissuing requests, trading off completeness for latency, and creating performance models to predict stragglers in the system.

A recurring pattern to reducing tail latency is to exploit the redundancy built into each tier of the application architecture. In this paper, we show that the problem of *replica selection* — wherein a *client* node has to make a choice about selecting one out of multiple *replica servers* to serve a request — is a first-order concern in this context. Interestingly, we find that the impact of the replica selection algorithm has often been overlooked. We argue that layering approaches like request duplication and reissues atop a poorly performing replica selection algorithm should be cause for concern. For example, reissuing requests but selecting poorly-performing nodes to process them increases system utilization [48] in exchange for limited benefits.

As we show in Section 2, the replica selection strategy has a direct effect on the tail of the latency distribution. This is particularly so in the context of data stores that rely on replication and partitioning for scalability, such as key-value stores. The performance of these systems is influenced by many sources of variability [16, 28] and running such systems in cloud environments, where utilization should be high and environmental uncertainty is a fact of life, further aggravates performance fluctuations [26].

Replica selection can compensate for these conditions by preferring faster replica servers whenever possible. However, this is made challenging by the fact that servers exhibit performance fluctuations over time. Hence, replica selection needs to quickly adapt to changing system dynamics. On the other hand, any reactive scheme in this context must avoid entering pathological behaviors that lead to load imbalance among nodes and oscillating instabilities. In addition, replica selection should not be computationally costly, nor require significant coordination overheads.

In this paper, we present C3, an adaptive replica selection mechanism that is robust in the face of fluctuations in system performance. At the core of C3's design, two key concepts allow it to reduce tail latencies and hence improve performance predictability. First, using simple and inexpensive feedback from servers, clients make use of a replica ranking function to prefer faster servers and compensate for slower service times, all while ensuring that the system does not enter herd behaviors or load-oscillations. Second, in C3, clients implement a distributed rate control mechanism to ensure that, even at high fan-ins, clients do not overwhelm individual servers. The combination of these mechanisms enable C3 to reduce queuing delays at servers while the system remains reactive to variations in service times.

Our study applies to any low-latency data store wherein replica diversity is available, such as a key-value store. We hence base our study on the widely-used [15] Cassandra distributed database [5], which is designed to store and serve larger-than-memory datasets. Cassandra powers a variety of applications at large web sites such as Netflix and eBay [6]. Compared to other related systems (Table 1), Cassandra implements a more sophisticated load-based replica selection mechanism as well, and is thus a better reference point for our study. However, C3 is applicable to other systems and environments that need to exploit replica diversity in the face of performance variability, such as a typical multi-tiered application or other data stores such as MongoDB or Riak.

In summary, we make the following contributions:

1. Through performance evaluations on Amazon EC2, we expose the fundamental challenges involved in managing tail latencies in the face of service-time variability (§2).
2. We develop an adaptive replica selection mechanism, C3, that reduces the latency tail in the presence of service-time fluctuations in the system. C3 does not make use of request reissues, and only relies on minimal and approximate information exchange between clients and servers (§3).

Cassandra	*Dynamic Snitching*: considers history of read latencies and I/O load
OpenStack Swift	Read from a single node and retry in case of failures
MongoDB	Optionally select nearest node by network latency (does not include CPU or I/O load)
Riak	Recommendation is to use an external load balancer such as Nginx [38]

Table 1: Replica selection mechanisms in popular NoSQL solutions. Only Cassandra employs a form of adaptive replica selection (§2.3).

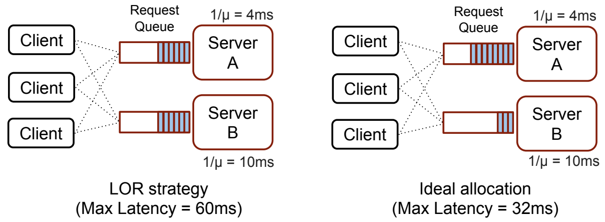
3. We implement C3 (§4) in the Cassandra distributed database and evaluate it through experiments conducted on Amazon EC2 (for accuracy) (§5) and simulations (for scale) (§6). We demonstrate that our solution improves Cassandra's latency profile along the mean, median, and the tail (by up to a factor of 3 at the 99.9th percentile) whilst improving read throughput by up to 50%.

2 The Challenge of Replica Selection

In this section, we first discuss the problem of time-varying performance variability in the context of cloud environments. We then underline the need for load-based replica selection schemes and the challenges associated with designing them.

2.1 Performance fluctuations are the norm

Servers in cloud environments routinely experience performance fluctuations due to a multitude of reasons. Citing experiences at Google, Dean and Barroso [16] list many sources of latency variability that occur in practice. Their list includes, but is not limited to, contention for shared resources within different parts of and between applications (further discussed in [26]), periodic garbage collection, maintenance activities (such as log compaction), and background daemons performing periodic tasks [40]. Recently, an experimental study of response times on Amazon EC2 [50] illustrated that long tails in latency distribution can also be exacerbated by virtualization. A study [23] of interactive services at Microsoft Bing found that over 30% of analyzed services have 95th percentile of latency 3 times their median latency. Their analysis showed that a major cause for the high service performance variability is that latency varies greatly across machines and time. Lastly, a common workflow involves accessing large volumes of data from a data store to serve as inputs for batch jobs on large-



LOR strategy (Max Latency = 60ms)

Ideal allocation (Max Latency = 32ms)

Figure 1: *Left*: how the least-outstanding requests (LOR) strategy allocates a burst of requests across two servers when executed individually by each client. *Right:* An ideal allocation that compensates for higher services time with lower queue lengths.

scale computing platforms such as Hadoop, and injecting results back into the data store [45]. These workloads can introduce latency spikes at the data store and further impact on end-user delays.

As part of our study, we spoke with engineers at Spotify and SoundCloud, two companies that use and operate large Cassandra clusters in production. Our discussions further confirmed that all of the above mentioned causes of performance variability are true pain points. Even in well provisioned clusters, unpredictable events such as garbage collection on individual hosts can lead to latency spikes. Furthermore, Cassandra nodes periodically perform *compaction*, wherein a node merges multiple SSTables [5, 13] (the on-disk representation of the stored data) to minimize the number of SSTable files to be consulted per-read, as well as to reclaim space. This leads to significantly increased I/O activity.

Given the presence of time-varying performance fluctuations, many of which can potentially occur even at sub-second timescales [16], it is important that systems gracefully adapt to changing conditions. By exploiting server redundancy in the system, we investigate how replica selection effectively reduces the tail latency.

2.2 Load-based replica selection is hard

Accommodating time-varying performance fluctuations across nodes in the system necessitates a replica selection strategy that takes into account the load across different servers in the system. A strategy commonly employed by many systems is the *least-outstanding requests* strategy (LOR). For each request, the client selects the server to which it has the least number of outstanding requests. This technique is simple to implement and does not require global system information, which may not be available or is difficult to obtain in a scalable fashion. In fact, this is commonly used in load-balancing applications such as Nginx [34] (recommended as a load-balancer for Riak [38]) or Amazon ELB [3].

However, we observe that this technique is not ideal for reducing the latency tail, especially since many realistic workloads are skewed in practice and access patterns change over time [9]. Consider the system in Figure 1, with two replica servers that at a particular point in time have service times of 4 ms and 10 ms respectively. Assume all three clients receive a burst of 4 requests each. Each request needs to be forwarded to a single server. Based on purely local information, if every client selects a server using the LOR strategy, it will result in each server receiving an equal share of requests. This leads to a maximum latency of 60 ms, whereas an ideal allocation in this case obtains a maximum latency of 32 ms. We note that *LOR* over time will prefer faster servers, but by virtue of purely relying on local information, it does not account for the existence of other clients with potentially bursty workloads and skewed access patterns, and does not explicitly adapt to fast-changing service times.

Designing distributed, adaptive and stable load-sensitive replica selection techniques is challenging. If not carefully designed, these techniques can suffer from "herd behavior" [32, 39]. Herd behavior leads to load oscillations, wherein multiple clients are coaxed to direct requests towards the least-loaded server, degrading the server's performance, which subsequently causes clients to repeat the same procedure with a different server.

Indeed, looking at the landscape of popular data stores (Table 1), we find that most systems only implement very simple schemes that have little or no ability to react quickly to service-time variations nor distribute requests in a load-sensitive fashion. Among the systems we studied, Cassandra implements a more sophisticated strategy called Dynamic Snitching that attempts to make replica selection decisions informed by histories of read latencies and I/O loads. However, through performance analysis of Cassandra, we find that this technique suffers from several weaknesses, which we discuss next.

2.3 Dynamic Snitching's weaknesses

Cassandra servers organize themselves into a one-hop distributed hash table. A client can contact any server for a read request. This server then acts as a *coordinator*, and internally fetches the record from the node hosting the data. Coordinators select the best replica for a given request using *Dynamic Snitching*. With Dynamic Snitching, every Cassandra server ranks and prefers faster replicas by factoring in read latencies to each of its peers, as well as I/O load information that each server shares with the cluster through a gossip protocol.

Given that Dynamic Snitching is load-based, we evaluate it to characterize how it manages tail-latencies and if it is subject to entering load-oscillations. Indeed, our experiments on Amazon EC2 with a 15-node Cassandra

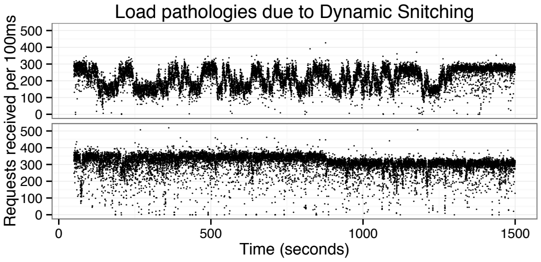


Figure 2: Example load oscillations seen by a given node in Cassandra due to Dynamic Snitching, in measurements obtained on Amazon EC2. The y-axis represents the number of requests processed in a 100 ms window by a Cassandra node. Even under stable conditions (*bottom*), the number of requests processed in a 100 ms window by a node ranges from 0 up to 500, which is symptomatic of herd behavior.

cluster confirm this (the details of the experimental setup are described in § 5). In particular, we recorded heavy-tailed latency characteristics wherein the difference between the 99.9^{th} percentile latencies are *up to 10 times* that of the median. Furthermore, we recorded the number of read requests individual Cassandra nodes serviced in 100 ms intervals. For every run, we observed the node that contributed most to the overall throughput. These nodes consistently exhibited synchronized load oscillations, example sequences of which are shown in Figure 2. Additionally, we confirmed our results with the Spotify engineers, who have also encountered load instabilities that arise due to garbage-collection induced performance fluctuations in the system [29].

A key reason for Dynamic Snitching's vulnerability to oscillations is that each Cassandra node re-computes scores for its peers at fixed, discrete intervals. This interval based scheme poses two problems. First, the system cannot react to time-varying performance fluctuations among peers that occur at time-scales less than the fixed-interval used for the score recomputation. Second, by virtue of fixing a choice over a discrete time interval (100 ms by default), the system risks synchronization as seen in Figure 2. While one may argue that this can be overcome by shortening the interval itself, the calculation performed to compute the scores is expensive, as it is also stated explicitly in the source code; a median over a history of exponentially weighted latency samples (that is reset only every 10 minutes) has to be computed for each node as part of the scoring process. Additionally, Dynamic Snitching relies on gossiping one second averages of `iowait` information between nodes to aid with the ranking procedure (the intuition being that nodes can avoid peers who are performing compaction). These `iowait` measurements influence the scores used for ranking peers heavily (up to two orders of magnitude more influence than latency measurements). Thus, an external or internal perturbation in I/O activity can influence a Cassandra node's replica selection loop for extended intervals. Together with the synchronization-prone behavior of having a periodically updated ranking, this can lead to poor replica selection decisions that degrade system performance.

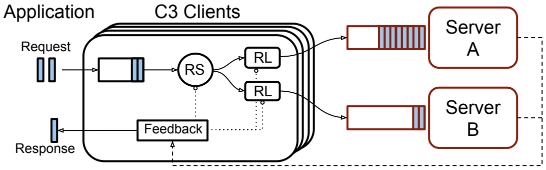


Figure 3: Overview of C3. RS: Replica Selection scheduler, RL: Rate Limiter of server $s \in [A, B]$.

3 C3 Design

C3 is an adaptive replica selection mechanism designed with the objective of reducing tail latency. Based on the considerations in Section 2, we design C3 while keeping in mind these two goals:

i) *Adaptive:* Replica selection must *cope and quickly react to heterogeneous and time-varying service times* across servers.

ii) *Well-behaved:* Clients performing replica selection must *avoid herd behaviors* where a large number of clients concentrate requests towards a fast server.

At the core of C3's design are the two following components that allow it to satisfy the above properties:

1. **Replica Ranking:** Using minimal and approximate feedback from individual servers, clients rank and prefer servers according to a scoring function. The scoring function factors in the existence of multiple clients and the subsequent risk of herd behavior, whilst allowing clients to prefer faster servers.
2. **Distributed Rate Control and Backpressure:** Every client rate limits requests destined to each server, adapting these rates in a fully-distributed manner using a congestion-control inspired technique [22]. When rate limits of all candidate servers for a request are exceeded, clients retain requests in a backlog queue until at least one server is within its rate limit again.

3.1 Replica ranking

With replica ranking, clients individually rank servers according to a scoring function, with the scores serving as a proxy for the latency to expect from the corresponding server. Clients then use these scores to prefer faster servers (lower scores) for each request. To reduce tail latency, we aim to minimize the product of queue-size (q_s)

and service-time ($1/\mu_s$, the inverse of the service rate) across every server s (Figure 1).

Delayed and approximate feedback. In C3, servers relay feedback about their respective q_s and $1/\mu_s$ on each response to a client. The q_s is recorded after the request has been serviced and the response is about to be dispatched. Clients maintain Exponentially Weighted Moving Averages (EWMA) of these metrics to smoothen the signal. We refer to these smoothed values as $\bar{q}_s$ and $\bar{\mu}_s$.

Accounting for uncertainty and concurrency. The delayed feedback from the servers lends clients only an approximate view of the load across the servers and is not sufficient by itself. Such a view is oblivious to the existence of other clients in the system, as well as the number of requests that are potentially in flight, and is thus prone to herd behaviors. It is therefore imperative that clients account for this potential concurrency in their estimation of each server's queue-size.

For each server s, a client maintains an instantaneous count of its outstanding requests os_s (requests for which a response is yet to be received). Clients calculate the queue-size estimate ($\hat{q}_s$) of each server as $\hat{q}_s = 1 + os_s \cdot w + \bar{q}_s$, where w is a weight parameter. We refer to the $os_s \cdot w$ term as the *concurrency compensation*.

The intuition behind the concurrency compensation term is that a client will always extrapolate the queue-size of a server by an estimated number of requests in flight. That is, it will always account for the possibility of multiple clients concurrently submitting requests to the same server. Furthermore, clients with a higher value of os_s will implicitly project a higher queue-size at s and thus rank it lower than a client that has sent fewer requests to s. Using this queue-size estimate to project the $\hat{q}_s/\bar{\mu}_s$ ratio results in a desirable effect: a client with a higher demand will be more likely to rank s poorly compared to a client with a lighter demand. This hence provides a degree of robustness to synchronization. In our experiments, we set w to the number of clients in the system. This serves as a good approximation in settings where the number of clients is comparable to the expected queue lengths at the servers.

Penalizing long queues. With the above estimation, clients can compute the $\hat{q}_s/\bar{\mu}_s$ ratio of each server and rank them accordingly. However, given the existence of multiple clients and time-varying service times, a function linear in $\hat{q}$ is not an effective scoring function for replica ranking. To see why, consider the example in Figure 4. The figure shows how clients would score two servers using a linear function: here, the service time estimates are 4 ms and 20 ms, respectively. We observe that under a linear scoring regime, for a queue-size estimate of 20 at the slower server, only a corresponding value of 100 at the faster server would cause a client to prefer the slower server again. If clients distribute requests by choosing the best replica according to this scoring function, they will build up and maintain long queues at the faster server in order to balance response times between the two nodes.

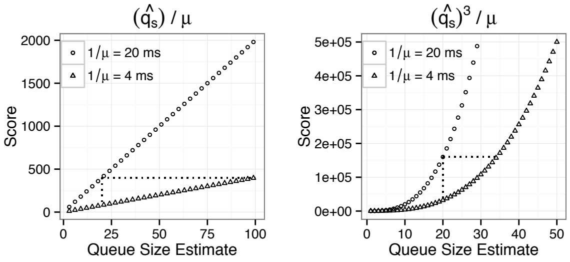


Figure 4: A comparison between linear (left) and cubic (right) scoring functions. For differing values of $1/\mu$, the difference in queue-size estimates required for the scores of two replicas to be equal is smaller for the cubic function (thus penalizing longer queues).

However, if the service time of the faster server increases due to an unpredictable event such as a garbage collection pause, *all requests* in its queue will incur higher waiting times. To alleviate this, C3's scoring function *penalizes longer queue lengths* using the same intuition behind that of delay costs as in [10,46]. That is, we use a non-decreasing convex function of the queue-size estimate in the scoring function to penalize longer queues. We achieve this by raising the $\hat{q}_s$ term in the scoring function to a higher degree, b: $(\hat{q}_s)^b/\bar{\mu}_s$.

Returning to the above example, this means the scoring function will treat the above two servers as being of equal score if the queue-size estimate of the faster server ($1/\mu = 4$ ms) is $\sqrt[b]{20/4}$ times that of the slower server ($1/\mu = 20$ ms). For higher values of b, clients will be less greedy about preferring a server with a lower μ^{-1}. We use $b = 3$ to have a cubic scoring function (Figure 4), which presents a good trade-off between clients preferring faster servers and providing enough robustness to time-varying service times.

Cubic replica selection. In summary, clients use the following scoring function for each replica:

$$\Psi_s = R_s - 1/\bar{\mu}_s + (\hat{q}_s)^3/\bar{\mu}_s$$

where $\hat{q}_s = 1 + os_s \cdot n + \bar{q}_s$ is the queue-size estimation term, os_s is the number of outstanding requests from the client to s, n is the number of clients in the system, and R_s, $\bar{q}_s$ and $\bar{\mu}_s{}^{-1}$ are EWMAs of the response time (as witnessed by the client),[1] queue-size and service time

[1] Note R_s implicitly accounts for network latency but we consider

feedback received from server s, respectively. The score reduces to R_s when the queue-size estimate term of the server is 1 (which can only occur if the client has no outstanding requests to s and the queue-size feedback is zero). Note that the $R_s - \mu_s^{-1}$ term's contribution to the score diminishes quickly when the client has a non-zero queue-size estimate (see Figure 4).

3.2 Rate control and backpressure

Replica selection allows clients to prefer faster servers. However, replica selection alone cannot ensure that the combined demands of all clients on a single server remain within that server's capacity. Exceeding capacity increases queuing on the server-side and reduces the system's reactivity to time-varying performance fluctuations. Thus, we introduce an element of rate-control to the system, wherein every client rate-limits requests to individual servers. If the rates of all candidate servers for a request are saturated, clients retain the request in a backlog queue until a server is within its rate limit again.

Decentralized rate control. To account for servers' performance fluctuations, clients need to adapt their estimations of a server's capacity and adjust their sending rates accordingly. As a design choice and inspired by the CUBIC congestion-control scheme [22], we opt to use a decentralized algorithm for clients to estimate and adapt rates across servers. That is, we avoid the need for clients to inform each other about their demands for individual servers, or for the servers to calculate allocations for potentially numerous clients individually. This further increases the robustness of our system; clients' adaptation to performance fluctuations in the system is not purely tied to explicit feedback from the servers.

Thus, every client maintains a token-bucket based rate-limiter for each server, which limits the number of requests sent to a server within a specified time window of δ ms. We refer to this limit as the *sending-rate* (*srate*). To adapt the rate limiter according to the perceived performance of the server, clients track the number of responses being received from a server in a δ ms interval, that is, the *receive-rate* (*rrate*). The rate-adaptation algorithm aims to adjust *srate* in order to match the *rrate* of the server.

Cubic rate adaptation function. Upon receiving a response from a server s, the client compares the current *srate* and *rrate* for s. If the client's sending rate is lower than the receive rate, it increases its rate according to a cubic function [22]:

$$srate \leftarrow \gamma \cdot \left(\Delta T - \sqrt[3]{\left(\frac{\beta \cdot R_0}{\gamma}\right)} \right)^3 + R_0$$

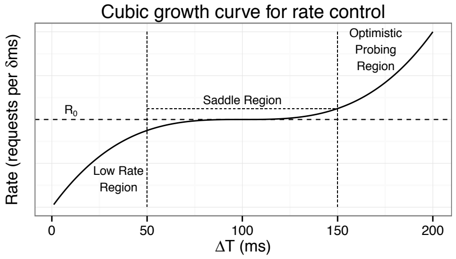


Figure 5: Cubic function for clients to adapt their sending rates

where ΔT is the elapsed time since the last rate-decrease event, and R_0 is the "saturation rate" — the rate at the time of the last rate-decrease event. If the receive-rate is lower than the sending-rate, the client decreases its sending-rate multiplicatively by β. γ represents a scaling factor and is chosen to set the desired duration of the saddle region (see § 4 for the values used).

Benefits of the cubic function. While we have not fully explored the vast design space for a rate adaptation technique, we were attracted to a cubic growth function because of its property of having a saddle region. The functioning of the *cubic* rate adaption strategy caters to the following three operational regions (Figure 5): (1) *Low-rates:* when the current sending rate is significantly lower than the saturation rate (after say, a multiplicative decrease),the client increases the rate steeply; (2) *Saddle region:* when the sending rate is close to the perceived saturation point of the server (R_0), the client stabilizes its sending rate, and increases it conservatively, and (3) *Optimistic probing:* if the client has spent enough time in the stable region, it will again increase its rate aggressively, and thus probe for more capacity. At any time, if the algorithm perceives itself to be exceeding the server's capacity, it will update its view of the server's saturation point and multiplicatively reduce its sending rate. The parameter γ can be adjusted for a desired length of the saddle region. Lastly, given that multiple clients may potentially be adjusting their rates simultaneously, for stability reasons, we cap the step size of a rate increase by a parameter $s_{\max}$.

3.3 Putting everything together

C3 combines distributed replica selection and rate control as indicated in Algorithms 1 and 2, with the control flow in the system depicted in Figure 3. When a request is issued at a client, it is directed to a replica selection scheduler. The scheduler uses the scoring function to order the subset of servers that can handle the request, that is, the replica group ($\mathscr{R}$). It then iterates through the list of replicas and selects the first server s that is within

that network congestion is not the source of performance fluctuations.

Algorithm 1 On Request Arrival (Request *req*, Replicas $\mathscr{R}$)

```
repeat
    $\mathscr{R} \leftarrow sort(\mathscr{R})$    ▷ sort replicas by cubic score function
    for Server $s$ in $\mathscr{R}$ do
        if $s$ within $srate_s$ then
            $consume\_token(srate_s)$
            $os_s \leftarrow os_s + 1$    ▷ update outstanding requests
            $send(req, s)$    ▷ send to server $s$
            return
    if $req$ not sent then
        wait until token available    ▷ Backpressure
until $req$ is sent
```

Algorithm 2 On Request Completion (Request *req*, Server *s*)

```
1:  $os_s \leftarrow os_s - 1$    ▷ update outstanding requests
2:  update EWMA of $q_s$, $\mu_s^{-1}$ feedback
3:  if $(srate_s > rrate_s$ && $now() - T_{inc} > hysteresis\_period)$ then
4:      $R_0 \leftarrow srate_s$
5:      $srate_s \leftarrow srate_s \cdot \beta$
6:      $T_{dec} \leftarrow now()$
7:  else if $(srate_s < rrate_s)$ then
8:      $\Delta T \leftarrow now() - T_{dec}$
9:      $T_{inc} \leftarrow now()$
10:     $R \leftarrow \gamma \cdot \left(\Delta T - \sqrt[3]{(\frac{\beta \cdot R_0}{\gamma})}\right)^3 + R_0$
11:     $srate_s \leftarrow \min(srate_s + s_{\max}, R)$
```

the rate as defined by the local rate limiter for s. If all replicas have exceeded their rate limits, the request is enqueued into a backlog queue. The scheduler then waits until at least one replica is within its rate before repeating the procedure. When a response for a request arrives, the client records the feedback metrics from the server and adjusts its sending rate for that server according to the cubic-rate adaptation mechanism. After a rate increase, a hysteresis period is enforced (Algorithm 2, line 3) before another rate-decrease so as to allow clients' receive-rate measurements enough time to catch up since the last increased sending rate at T_{inc}.

4 Implementation

We implemented C3 within Cassandra. For Cassandra's internal read-request routing mechanism, this means that every Cassandra node is both a C3 client and server (specifically, coordinators in Cassandra's read path are C3 clients). In vanilla Cassandra, every read request follows a synchronous chain of steps leading up to an eventual enqueuing of the request into a per-node TCP connection buffer. For C3, we modified this chain of steps to control the number of requests that would be pushed to the TCP buffers of each node. Recall that C3's replica scoring and rate control operate at the granularity of replica groups. Given that in Cassandra, there are as many replica groups as nodes themselves, we need as many backpressure queues and replica selection schedulers as there are nodes. Thus, every read-request upon arrival in the system needs to be asynchronously routed to a scheduler corresponding to the request's replica group. Lastly, when a coordinator node performs a remote read, the server that handles the request tracks the service time of the operation and the number of pending read requests in the server. This information is piggybacked to the coordinator and serves as the feedback for the replica ranking.

There are challenges in making this implementation efficient. For one, since a single remote peer can be part of multiple replica sets, multiple admission control schedulers may potentially contend to push a request from their respective backpressure queues towards the same endpoint. Care needs to be exercised that this does not lead to starvation. To handle this complexity, we relied upon the Akka framework [1] for message-passing concurrency (*Actor* based programming). With Akka, every per-replica-group scheduler is represented as a single actor, and we configured the underlying Java thread dispatcher to fair schedule between the actors. This design of having multiple backpressure queues also increases robustness, as one replica group entering backpressure will not affect other replica groups. The message queue that backs each Akka actor implicitly serves as the backpressure queue per-replica group. At roughly 600 bytes of overhead per actor, our extensions to Cassandra is thus lightweight. Our implementation amounted to 398 lines of code.[2]

For the rest of our study, we set the cubic rate adaptation parameters as follows: the multiplicative decrease parameter β is set to 0.2, and we configured γ to set the saddle region to be 100 ms long. We define the rate for each server as a number of permissible requests per 20 ms (δ), and use a hysteresis duration equal to twice the rate interval. We cap the cubic-rate step size (s_{max}) to 10. We did not conduct an exhaustive sensitivity analysis of all system parameters, which we leave for future work. Lastly, Cassandra uses read-repairs for anti-entropy; a fraction of read requests will go to all replicas (10% by default). This further allows coordinators to update their view of their peers.

5 System Evaluation

We evaluated C3 on Amazon EC2. Our Cassandra deployment comprised 15 m1.xlarge instances. We tuned the instances and Cassandra according to the officially recommended production settings from Datastax [12] as well as in consultation with our contacts from the industry who operate production Cassandra clusters.

On each instance, we configured a single RAID0 array encompassing the four ephemeral disks which served

[2] Based on a Cassandra 2.0 development version.

as Cassandra's data folder (we also experimented on instances with SSD storage as we report on later). As we don't have production workloads, we used the industry-standard Yahoo Cloud Serving Benchmark (YCSB) [14] to generate datasets and run our workloads while stressing Cassandra up to its maximum attainable throughput. We assign tokens to each Cassandra node such that nodes own equal segments of the keyspace. Cassandra's replication factor was set to 3. We inserted 500 million 1KB size records generated by YCSB, which served as the dataset. The workload against the cluster was driven from three instances of YCSB running in separate VMs, each running 40 request generators, for a total of 120 generators. Each generator has a TCP connection of its own to the Cassandra cluster. Generators create requests for keys distributed according to a Zipfian access pattern prescribed by YCSB, with Zipf parameter $\rho = 0.99$, drawing from a set of 10 million keys. We used three common workload patterns for Cassandra deployments to evaluate our scheme: read-heavy (95% reads – 5% writes), update-heavy (50% reads – 50% writes) and read-only (100% read). These workloads generate access patterns typical of photo tagging, session-store and user-profile applications, respectively [14]. The read and update heavy workloads in particular are popular across a variety of Cassandra deployments [18, 25]. Each measurement involves 10 million operations of the workload, and is repeated five times. Bar plots represent averages and 95th percentile confidence intervals.

In evaluating C3, we are interested in answering the following questions across various conditions:

1. Does C3 improve the tail latency without sacrificing the mean or median?
2. Does C3 improve the read throughput (requests/s)?
3. How well does C3 load condition the cluster and adapt to dynamic changes in the environment?

Impact of workload on latency: Figure 6 indicates the read latency characteristics of Cassandra across different workloads when using C3 compared to Dynamic Snitching (DS). Regardless of the workload used, C3 improves the latency across all the considered metrics, namely, the mean, median, 99^{th} and 99.9^{th} percentile latencies. Since the ephemeral storage in our instances are backed by spinning-head disks, the latency increases with the amount of random disk seeks. This explains why the read-heavy workload results in lower latencies than the read-only workload (since the latter causes more random seeks). Furthermore, C3 effectively shortens the ratio of tail-latencies to the median, leading to a more predictable latency profile. With the read-heavy workload, the difference between the 99.9^{th} percentile latency and

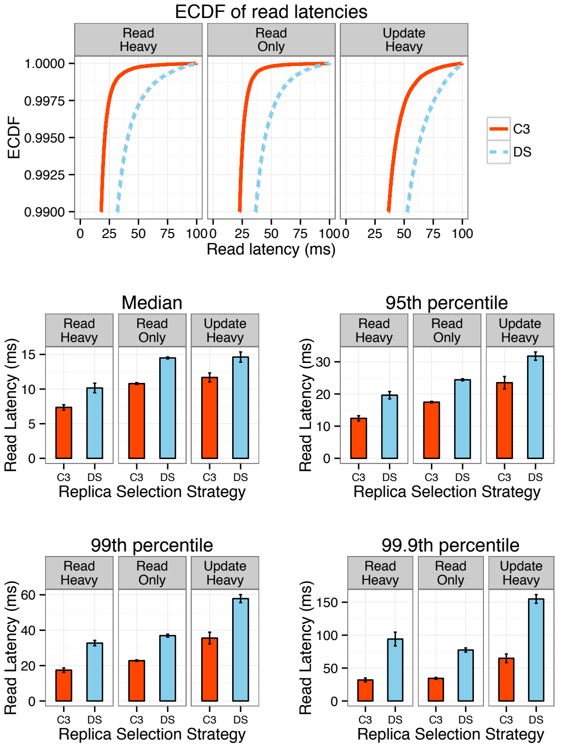


Figure 6: Cassandra's latency characteristics when using Dynamic Snitching and C3. C3 significantly improves the tail latency under different workloads without compromising the median.

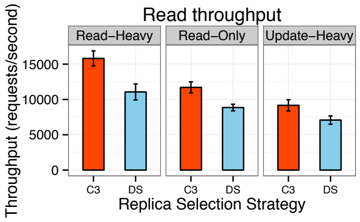


Figure 7: Throughput obtained with C3 and with Dynamic Snitching. C3 achieves higher throughput by better utilizing the available system capacity across replica servers.

the median is 24.5 ms with C3, whereas with DS, it is 83.91 ms: *more than 3x improvement*. In the update-heavy and read-only scenarios, C3 improves the same difference by a factor of 2.6 each. Besides the different percentiles, C3 also improves the mean latency by between 3 ms and 4 ms across all scenarios.

Impact of workload on read throughput: Figure 7 indicates the measured throughputs for C3 versus DS. By virtue of controlling waiting times across the replicas, C3 makes better use of the available system capacity, resulting in an increase in throughput across the considered workloads. In particular, C3 improves the throughput by

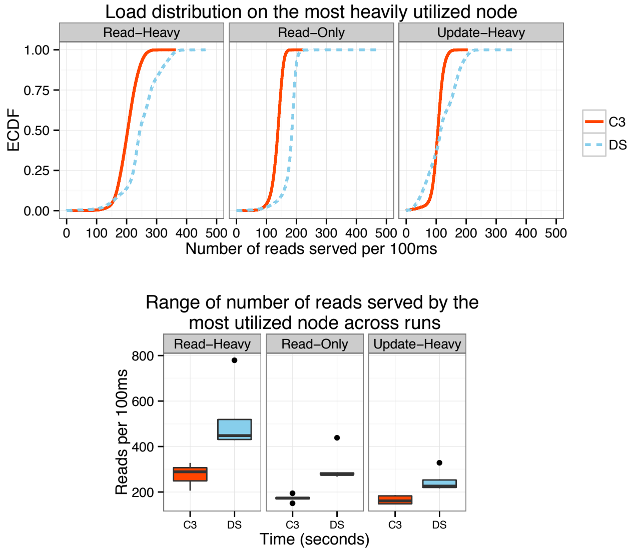


Figure 8: Aggregated distribution of number of reads serviced per 100 ms, by the most heavily loaded Cassandra node per run. With C3, the most heavily utilized node has a lower range in the load over time, wherein the difference between the 99th percentile and median number of requests served in 100 ms is lower than with Dynamic Snitching.

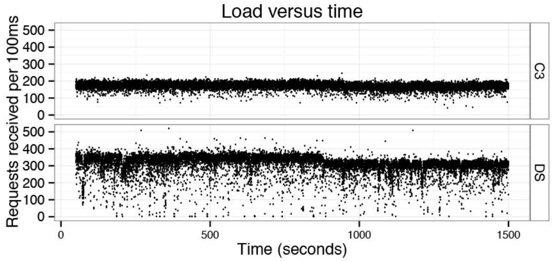


Figure 9: Example number of reads received by a single Cassandra node, per 100ms. With C3 (top), Cassandra coordinators internally adjust sending rates to match their peers' perceived capacity, leading to a smoother load profile free of oscillations. The per-server load is lower in C3 also because the requests are spread over more servers compared to DS (bottom).

between 26% and 43% across the considered workloads (update-heavy and read-heavy workloads respectively). We also note that the difference in throughput between the read- and update-heavy workloads of roughly 75% (across both strategies) is consistent with publicly available Cassandra benchmark data [18].

Impact of workload on load-conditioning: We now verify whether C3 fulfills its design objective of avoiding load pathologies. Since the key access pattern of our workloads are Zipfian distributed, we observe the load over time of the node that has served the highest number of reads across each run, that is, the most heavily utilized node. Figure 8 represents the distribution of the number of reads served per 100 ms by the most heavily utilized node in the cluster across runs. Note that *despite improving the overall system throughput*, the most heavily utilized node in C3 serves fewer requests than with DS. As a further confirmation of this, we present an example load profile as produced by C3 on highly utilized nodes (Figure 9). Unlike with DS, we do not see synchronized load-spikes when using C3, evidenced by the lack of oscillations and synchronized vertical bursts in the time-series. Furthermore, given that C3's rate control absorbs and distributes bursts carefully, it leads to a smoother load-profile wherein samples of the load in a given interval are closer to the system's true capacity unlike with DS.

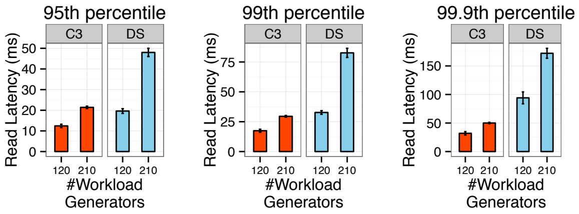


Figure 10: Overall performance degradation when increasing the number of workload generators from 120 to 210.

Performance at higher system utilization: We now compare C3 with DS to understand how the performance of both systems degrade with an increase in overall system utilization. We increase the number of workload generators from 120 to 210 (an increase of 75%). Figure 10 presents the tail latencies observed for the read-heavy workload. For a 75% increase in the demand, we observe that C3's latency profile, even at the 99.9^{th} percentile, degrades proportionally to the increase in system load. With DS, the median and 99.9^{th} percentile latencies degrade by roughly 82%, whereas the 95^{th} and 99^{th} percentile latencies *degrade by factors of up to 150%*. Furthermore, the mean latency with Dynamic Snitching is *70% higher* than with C3 under the higher load.

Adaptation to dynamic workload change: We now evaluate a scenario wherein an update-heavy workload enters a system where a read-heavy workload is already active, and observe the effect on the latter's read latencies. The experiment begins with 80 generators running a read-heavy workload against the cluster. After 640 s, an additional 40 generators enter the system, issuing update-heavy workloads. We observe the latencies from the perspective of the read-heavy generators around the 640 s mark. Figure 11 indicates a time-series of the latencies contrasting C3 versus DS. Each plot represents

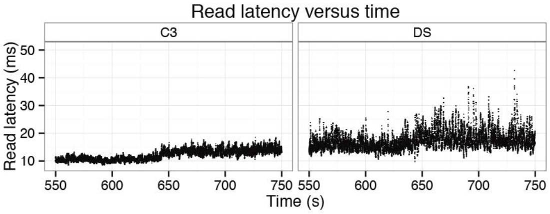


Figure 11: Dynamic workload experiment. The moving median over the latencies observed by the read-heavy generators from a run each involving C3 (left) and DS (right). At time 640 s, 40 new generators join the system and issue update-heavy workloads. With C3, the latencies degrade gracefully, whereas DS fails to avoid latency spikes.

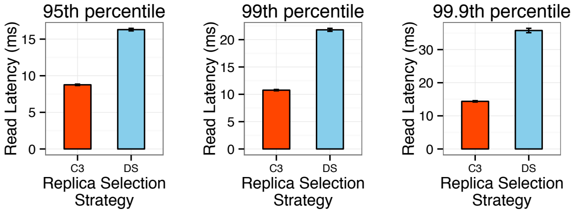


Figure 12: Results when using SSDs instead of spinning-head disks.

a 50-sample wide moving median[3] over the recorded latencies. Both DS and C3 react to the new generators entering the system, with a degradation of the read latencies observed at the 640 s mark. However, in contrast to DS, C3's latency profile degrades gracefully, evidenced by the lack of synchronized spikes visible in the time-series as is the case with DS.

Skewed record sizes: So far, we considered fixed-length records. Since C3 relies on per-request feedback of the service times in the system, we observe whether variable length records may introduce any anomalies in the control loop. We use YCSB to generate a similar dataset as before, but where field sizes are Zipfian distributed (favoring shorter values). The maximum record length is 2KB, with each record comprising the key, and ten fields. Again, C3 improves over DS along all the considered latency metrics. In particular, with C3, the 99th percentile latency is just under 14 ms, whereas that of DS is close to 30 ms; *more than 2x improvement*.

Performance when using SSDs: As a further demonstration of C3's generality, we also perform measurements with m3.xlarge instances, which are backed by two 40 GB SSD disks. We configured a RAID0 array encompassing both disks. We reduced the dataset size to 150 million 1KB records in order to ensure that the dataset fits the reduced disk capacities of all nodes. Given that with SSDs, the system can sustain a higher workload, we used 210 read-heavy generators (70 threads per YCSB instance). Figure 12 illustrates the latency improvements obtained when using C3 versus DS with SSD backed instances. Even under the higher load, both algorithms have significantly lower latencies than when using spinning head disks. However, C3 again improves the 99.9th percentile latency by *more than 3x*. Furthermore, the difference between the 99th and 99.9th percentile latencies in C3 is *under 5 ms*, whereas with DS, it is on the order of 20 ms. Lastly, C3 also improves the average latency by roughly 3 ms, and increases the read throughput by 50% of that obtained by DS.

Comparison against request reissues: Cassandra has an implementation of speculative retries [16] as a means of reducing tail latencies. After sending a read request to a replica, the coordinator waits for the response until a configurable duration before reissuing the request to another replica. We evaluated the performance of DS with speculative retries, configured to fire after waiting until the 99th percentile latency. However, we observed that latencies actually degraded significantly after making use of this feature, up to a factor of 5 at the 99th percentile. We attribute this to the following cause: in the presence of highly variable response times across the cluster (already due to DS), coordinators potentially speculate too many requests. This increases the load on disks, further increasing seek latencies. Due to this anomaly, we did not perform further experiments. We however, leave a note of caution that speculative retries are not a silver bullet when operating a system at high utilization [48].

Sending rate adaptation and backpressure over time: Lastly, we turn to a seven-node Cassandra cluster in our local testbed to depict how nodes adapt their sending rates over time. Figure 13 presents a trace of the sending rate adaptation performed by two coordinators against a third node (*tracked node*). During the run, we artificially inflated the latencies of the tracked node thrice (using the Linux `tc` utility), indicated by the drops in throughput in the interval (45, 55) s, as well as the two shorter drops at times 59 s and 67 s. Observe that both coordinators' estimations of their peer's capacity agree over time. Furthermore, the figure depicts all three rate regimes of the cubic rate control mechanism. The points close to 1 on the y-axis are arrived at via the multiplicative decrease, causing the system to enter the low-rate regime. At that point, C3 aggressively increases its rate to be closer to the tracked saturation rate, entering the saddle region (along the smoothened median). The stray points above the smoothened median are points where C3 optimistically probes for more capacity. During this run,

[3] A moving median is better suited to reveal the underlying trend of a high-variance time-series than a moving average [7]

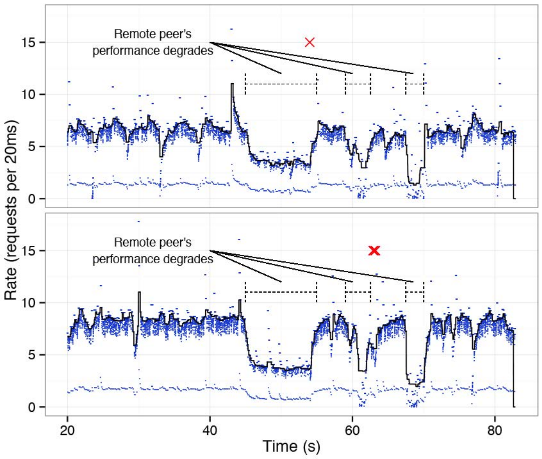


Figure 13: Sending rate adaptation performed by two coordinators against a third server. The receiving server's latency is artificially inflated thrice. The blue dots represent the sending-rates as adjusted by the cubic rate control algorithm, the black line indicates a moving median of the sending rates, and the red X marks indicate moments when affected replica group schedulers enter backpressure mode.

the backpressure mechanism fired 4 times (3 of which are very close in time) across both depicted coordinator nodes. Recall that backpressure is exerted when *all* replicas of a replica group have exceeded their rate limits. When the tracked node's latencies are reset to normal, the YCSB generators throttle up, sending a heavy burst in a short time interval. This causes a momentary surge of traffic towards the tracked node, forcing the corresponding replica selection schedulers to apply backpressure.

6 Evaluation Using Simulations

We turn to simulations to further evaluate C3 under different scenarios. Our objective is to study the C3 scheme independently of the intricacies of Cassandra and draw more general results. Furthermore, we are interested in understanding how the scheme performs under different operational extremes. In particular, we explore how C3's performance varies according to *(i)* different frequencies of service time fluctuations, *(ii)* lower utilization levels, and *(iii)* under skewed client demands.

Experimental setup: We built a discrete-event simulator[4], wherein workload generators create requests at a set of clients, and the clients then use a replica selection algorithm to route requests to a set of servers. A request generated at a client has a uniform probability of being forwarded to any replica group (that is, we do not model keys being distributed across servers according to consistent hashing as in Cassandra). The workload generators create requests according to a Poisson arrival process, to mimic arrival of user requests at web servers [35]. Each server maintains a FIFO request queue. To model concurrent processing of requests, each server can service a tunable number of requests in parallel (4 in our settings). The service time each request experiences is drawn from an exponential distribution (as in [48]) with a mean service time $\mu^{-1} = 4$ ms. We incorporated time-varying performance fluctuations into the system as follows: every T ms (*fluctuation interval*), each server, independently and with a uniform probability, sets its service rate either to μ or to $\mu \cdot D$, where D is a range parameter (thus, a bimodal distribution for server performance [41]). We set the D parameter to 3 (qualitatively, our results apply across multiple tested values of D, which we omit for brevity). The request arrival rate corresponds to 70% (high utilization scenario) and 45% (low utilization scenario) of the average service rate of the system, considering the time-varying nature of the servers' performance (that is, as if the service rate of each server's processor was $(\mu + D \cdot \mu)/2$). As with our experiments using Cassandra, we use a read-repair probability of 10% and a replication factor of 3, which further increases the load on the system. We use 200 workload generators, 50 servers, and vary the number of clients from 150 to 300. We set the one-way network latency to 250 μs. We repeat every experiment 5 times using different random seeds. 600,000 requests are generated in each run.

We compare C3 against three strategies:

1. **Oracle (ORA):** each client chooses based on perfect knowledge of the instantaneous q/μ ratio of the replicas (no required feedback from servers).
2. **Least-Outstanding Requests (LOR):** each client selects a replica to which it has sent the least number of requests so far.
3. **Round-Robin (RR):** as in C3, each client maintains a per-replica rate limiter. However, here it uses a round-robin scheme to allocate requests to replicas in place of C3's replica ranking. This allows us to evaluate the contribution of just rate limiting to the effectiveness of C3.

We also ran simulations of strategies such as uniform random, least-response time, and different variations of weighted random strategies. These strategies did not fare well compared to *LOR*, and due to space limits, we do not present results for them. We do not model disk activity in the simulator, and thus avoid comparing against Dynamic Snitching (since it relies on gossiping disk `iowait` measurements).

Impact of time-varying service times: Given that C3 clients rely on feedback from servers, we study the effect

[4] Code at `https://github.com/lalithsuresh/absim`.

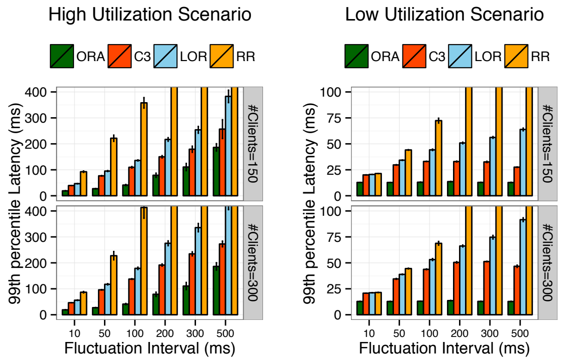


Figure 14: Impact of time-varying service times at high-utilization and low-utilization scenarios. Bars exceeding 400 ms are not shown.

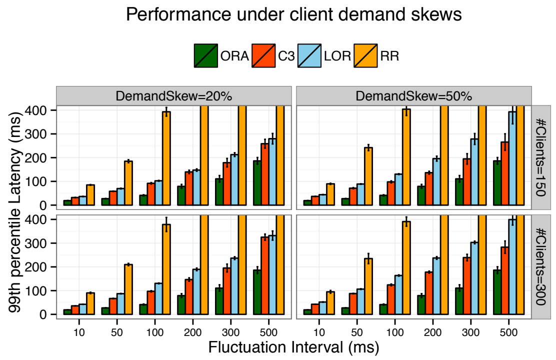


Figure 15: Impact of demand skews: 20% and 50% of the clients generate 80% of the requests to the servers. Bars exceeding 400 ms are not shown.

of the service-time fluctuation frequency on C3's control over the tail latency. Figure 14 presents the 99^{th} percentile tail-latencies when using C3 with 150 and 300 clients. When the average service times of the servers in the system change every 10 ms, C3 performs similarly to *LOR* and *RR*. This is expected, because at such a high frequency of performance variability, clients can make use of one round-trip's worth of feedback for at most another request, before that information is stale. However, as the interval between service-time changes increases, *LOR*'s performance degrades more compared to that of C3. Furthermore, the performance of *RR* suggests that rate-limiting alone does not improve the latency tail. This is because *RR* does not proactively prefer faster servers.. We also note that C3's performance remains relatively close to that of the *ORA*.

Performance at low utilization: While C3 is geared towards high-utilization environments with a number of requests in flight [35, 42], we now demonstrate the efficacy of C3 under low-utilization settings as well. We set the arrival rate to match a 45% system utilization. While the performances of *LOR* and *RR* degrade with higher fluctuation intervals, C3's performance begins to plateau instead. This is because a client using *LOR*, will allocate requests to slow servers as long as it has assigned more requests to other replicas. This leads to poor allocations as initially explained in Figure 1. Thus, the longer a server remains bad, the higher the chance that it will receive some requests when clients use the *LOR* strategy. On the other hand, C3 explicitly aims to equalize the product of the queue-size and μ^{-1} across servers, and thus does not use slow servers any more than required to balance the latency distribution. We reiterate that while the average service times of a server may change more slowly, the response times are still subject to the dynamic waiting times as a result of queuing at the server.

Performance under heavy demand-skews: Lastly, we study the effect of heavy demand skews on the observed latencies. Figure 15 presents results when 20% and 50% of C3 clients generate 80% of the total demand towards the servers, respectively. Again, regardless of the demand skew, C3 outperforms *LOR* and *RR*.

7 Discussion

How general is C3? C3 combines two mechanisms in order to carefully manage tail latencies in a distributed system: ***(i)*** a load-balancing scheme that is informed by a continuous stream of in-band feedback about a server's load, and ***(ii)*** distributed rate-control and backpressure. We believe that the ideas discussed here can be applied to any low-latency data store that can benefit from replica diversity. Furthermore, our simulations compared C3 against different replica selection mechanisms, and allowed us to decouple the workings of the algorithms themselves from the intricacies of running them within a complex system such as Cassandra. That said, we are currently porting C3 onto systems such as MongoDB and token-aware Cassandra clients such as Astyanax [8] (which will avoid the problem of clients selecting overloaded coordinators).

Long-term versus short-term adaptations: A common recommended practice among operators is to overprovision distributed systems deployed on cloud platforms in order to accommodate performance variability [37]. Unlike application servers, storage nodes that handle larger-than-memory datasets are not easily scaled up or down; adding a new node to the cluster and the subsequent re-balancing of data are operations that happen over timescales of hours. Such questions of provisioning sufficient capacity for a demand is orthogonal to our work; our objective with C3 is to carefully utilize *already provisioned* system resources in the face of performance variability over short timescales.

Strongly consistent reads: Our work has focused on selecting one out of a given set of replicas, which inherently assumes eventual consistency. This applies to common use-cases at large web-services today, including Facebook's accesses to its social graph [47], and most of Netflix's Cassandra usage [24]. However, it remains to be seen how our work can be applied to strongly consistent reads as well. In particular, the gains in such a scenario depend on the synchronization overhead of the respective read protocol, and the effect of a straggler cannot be easily avoided.

8 Related Work

Dean and Barroso [16] described techniques employed at Google to tolerate latency variability. They discuss short-term adaptations in the form of request reissues, along with additional logic to support preemption of duplicate requests to reduce unacceptable additional load. In D-SPTF [30], a request is forwarded to a single server. If the server has the data in its cache, it will respond to the query. Otherwise, the server forwards the request to all replicas, which then make use of cross-server cancellations to reduce load as in [16]. Vulimiri *et al.* [48] also make use of duplicate requests. They formalize the different threshold points at which using redundancy aids in minimizing the tail. In the context of Microsoft Azure and Amazon S3, CosTLO [49] also presents the efficacy of duplicate requests in coping with performance variability. In contrast to these works, our approach does not rely on redundant requests and is in essence complementary to the above in that request reissues could be introduced atop C3.

Kwiken [23] decomposes the problem of minimizing end-to-end latency over a processing DAG into a manageable optimization over individual stages, wherein the latency reduction techniques (e.g., request reissues) are complementary to our approach.

Pisces [42] is a multi-tenant key-value store architecture that provides fairness guarantees between tenants. It is concerned with fair-sharing the data-store and presenting proportional performances to different tenants. PARDA [21] is also focused on the problem of sharing storage bandwidth according to proportional-share fairness. Stout [31] uses congestion control to respond to to storage-layer performance variability by adaptively batching requests. PriorityMeister [53] focuses on providing tail latency QoS for bursty workloads in shared networked storage by combining priorities and rate limiters. As in C3, these works make use of TCP-inspired congestion control techniques for allocating storage resources across clients. While orthogonal to the problem of replica selection, we are planning to investigate the ideas embodied in these works within the context of C3. Pisces recognizes the problem of weighted replica selection but employs a round robin algorithm similar to the one used in our simulation results.

Mitzenmacher [33] showed that allowing a client to choose between two randomly selected servers based on queue lengths exponentially improves load-balancing performance over a uniform random scheme. This approach is embodied within systems such as Sparrow [36]. However, in our settings, replication factors are typically small compared to cluster size. Given a common replication factor of 3, ranking 3 servers instead of 2 only incurs a negligible overhead. Moreover, the basic power of two choices strategy does not include a rate limiting component to avoid exceeding server capacities, in contrast to C3. A thorough comparison between the two approaches is left for future work.

Lastly, there is much work in the cluster computing space on skew-tolerance [4, 19, 27, 44, 51]. In contrast to our work, cluster jobs operate at timescales of at least a few hundreds of milliseconds [36], if not minutes or hours.

9 Conclusion

In this paper, we highlighted the challenges involved in making a replica selection scheme explicitly cope with performance fluctuations in the system and environment. We presented the design and implementation of C3. C3 uses a combination of in-band feedback from servers to rank and prefer faster replicas along with distributed rate control and backpressure in order to reduce tail latencies in the presence of service-time fluctuations. Through comprehensive performance evaluations, we demonstrate that C3 improves Cassandra's mean, median and tail latencies (by up to 3 times at the 99.9^{th} percentile), all while increasing read throughput and avoiding load pathologies.

Acknowledgments: We thank our shepherd Lidong Zhou and the anonymous reviewers for their feedback. We are grateful to Sean Braithwaite, Conor Hennessy, Axel Liljencrantz, Jimmy Mårdell, and Radovan Zvonček for their valuable inputs and suggestions. This paper benefited greatly from discussions with Ramin Khalili, Georgios Smaragdakis and Olivier Bonaventure. This research is (in part) supported by European Union's Seventh Framework Program FP7/2007-2013 under the BigFoot project (grant agreement 317858) and the Trilogy2 project (grant agreement 317756), and by an AWS in Education Grant award.

References

[1] Akka. http://akka.io/, accessed Sept 25, 2014.

[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] Amazon ELB. http://docs.aws.amazon.com/ElasticLoadBalancing/latest/DeveloperGuide/TerminologyandKeyConcepts.html, accessed Sept 24, 2014.

[4] G. Ananthanarayanan, S. Kandula, A. Greenberg, I. Stoica, Y. Lu, B. Saha, and E. Harris. Reining in the Outliers in Map-reduce Clusters Using Mantri. In *OSDI*, 2010.

[5] Apache Cassandra. http://cassandra.apache.org/, accessed June 10, 2013.

[6] Apache Cassandra Use Cases. http://planetcassandra.org/apache-cassandra-use-cases/, accessed Sept 25, 2014.

[7] G. R. Arce. *Nonlinear Signal Processing: A Statistical Approach*. Wiley, 2004.

[8] Astyanax. https://github.com/Netflix/astyanax, accessed Jan 5, 2015.

[9] B. Atikoglu, Y. Xu, E. Frachtenberg, S. Jiang, and M. Paleczny. Workload Analysis of a Large-scale Key-value Store. In *SIGMETRICS*, 2012.

[10] C. F. Bispo. The single-server scheduling problem with convex costs. *Queueing Systems*, 73(3), 2013.

[11] J. Brutlag. Speed Matters, accessed Sept 24, 2014. http://googleresearch.blogspot.com/2009/06/speed-matters.html.

[12] Cassandra Documentation. http://www.datastax.com/documentation/cassandra/2.0, accessed Sept 25, 2014.

[13] F. Chang, J. Dean, S. Ghemawat, W. C. Hsieh, D. A. Wallach, M. Burrows, T. Chandra, A. Fikes, and R. E. Gruber. Bigtable: A Distributed Storage System for Structured Data. *ACM Trans. Comput. Syst.*, 26(2), June 2008.

[14] B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears. Benchmarking Cloud Serving Systems with YCSB. In *SoCC*, 2010.

[15] DB-Engines Ranking of Wide Column Stores. http://db-engines.com/en/ranking/wide+column+store, accessed Sept 25, 2014.

[16] J. Dean and L. A. Barroso. The Tail At Scale. *Communications of the ACM*, 56:74–80, 2013.

[17] G. Decandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, and W. Vogels. Dynamo: Amazon's Highly Available Key-value Store. In *SOSP*, 2007.

[18] J. Ellis. How not to benchmark Cassandra: a case study, 2014. http://www.datastax.com/dev/blog/how-not-to-benchmark-cassandra-a-case-study.

[19] A. D. Ferguson, P. Bodik, S. Kandula, E. Boutin, and R. Fonseca. Jockey: Guaranteed Job Latency in Data Parallel Clusters. In *EuroSys*, 2012.

[20] W. D. Gray and D. Boehm-Davis. Milliseconds matter: An introduction to microstrategies and to their use in describing and predicting interactive behavior. *Journal of Experimental Psychology: Applied*, 6, 2000.

[21] A. Gulati, I. Ahmad, and C. A. Waldspurger. PARDA: Proportional Allocation of Resources for Distributed Storage Access. In *FAST*, 2009.

[22] S. Ha, I. Rhee, and L. Xu. CUBIC: A New TCP-Friendly High-Speed TCP Variant. *SIGOPS Oper. Syst. Rev.*, 42(5), 2008.

[23] V. Jalaparti, P. Bodik, S. Kandula, I. Menache, M. Rybalkin, and C. Yan. Speeding up Distributed Request-Response Workflows. In *SIGCOMM*, 2013.

[24] C. Kalantzis. Eventual Consistency != Hopeful Consistency, talk at Cassandra Summit, 2013. https://www.youtube.com/watch?v=A6qzx_HE3EU.

[25] C. Kalantzis. Revisiting 1 Million Writes per second, 2014. http://techblog.netflix.com/2014/07/revisiting-1-million-writes-per-second.html.

[26] M. Kambadur, T. Moseley, R. Hank, and M. A. Kim. Measuring Interference Between Live Datacenter Applications. In *SC*, 2012.

[27] R. Kapoor, G. Porter, M. Tewari, G. M. Voelker, and A. Vahdat. Chronos: Predictable Low Latency for Data Center Applications. In *SoCC*, 2012.

[28] J. Li, N. K. Sharma, D. R. K. Ports, and S. D. Gribble. Tales of the Tail: Hardware, OS, and Application-level Sources of Tail Latency. In *SoCC*, 2014.

[29] A. Liljencrantz. How Not to Use Cassandra, talk at Cassandra Summit, 2013. https://www.youtube.com/watch?v=0u-EKJBPrj8.

[30] C. R. Lumb and R. Golding. D-SPTF: Decentralized Request Distribution in Brick-based Storage Systems. *SIGOPS Oper. Syst. Rev.*, 38(5):37–47, Oct. 2004.

[31] J. C. McCullough, J. Dunagan, A. Wolman, and A. C. Snoeren. Stout: An Adaptive Interface to Scalable Cloud Storage. In *USENIX ATC*, 2010.

[32] M. Mitzenmacher. How Useful Is Old Information? *IEEE Trans. Parallel Distrib. Syst.*, 11(1), Jan. 2000.

[33] M. Mitzenmacher. The power of two choices in randomized load balancing. *IEEE Trans. Parallel Distrib. Syst.*, 12(10), Oct. 2001.

[34] Nginx. http://nginx.org/en/docs/http/load_balancing.html, accessed Sept 24, 2014.

[35] R. Nishtala, H. Fugal, S. Grimm, M. Kwiatkowski, H. Lee, H. C. Li, R. McElroy, M. Paleczny, D. Peek, P. Saab, D. Stafford, T. Tung, and V. Venkataramani. Scaling Memcache at Facebook. In *NSDI*, 2013.

[36] K. Ousterhout, P. Wendell, M. Zaharia, and I. Stoica. Sparrow: Distributed, Low Latency Scheduling. In *SOSP*, 2013.

[37] Riak. AWS Performance Tuning, accessed Sept 24, 2014. http://docs.basho.com/riak/latest/ops/tuning/aws/.

[38] Riak. Load Balancing and Proxy Configuration, accessed Sept 24, 2014. http://docs.basho.com/riak/1.4.0/cookbooks/Load-Balancing-and-Proxy-Configuration/.

[39] M. Roussopoulos and M. Baker. Practical Load Balancing for Content Requests in Peer-to-Peer Networks. *Distributed Computing*, 18(6), 2006.

[40] S. Sanfilippo. Redis latency spikes and the 99th percentile, 2014. http://antirez.com/news/83.

[41] J. Schad, J. Dittrich, and J.-A. Quiané-Ruiz. Runtime Measurements in the Cloud: Observing, Analyzing, and Reducing Variance. *VLDB Endowment*, 3(1-2), Sept. 2010.

[42] D. Shue, M. J. Freedman, and A. Shaikh. Performance Isolation and Fairness for Multi-tenant Cloud Storage. In *OSDI*, 2012.

[43] S. Souders. Velocity and the Bottom Line, 2009. `http://radar.oreilly.com/2009/07/velocity-making-your-site-fast.html`.

[44] C. Stewart, A. Chakrabarti, and R. Griffith. Zoolander: Efficiently Meeting Very Strict, Low-Latency SLOs. In *USENIX ICAC*, 2013.

[45] R. Sumbaly, J. Kreps, L. Gao, A. Feinberg, C. Soman, and S. Shah. Serving Large-scale Batch Computed Data with Project Voldemort. In *FAST*, 2012.

[46] J. A. van Mieghem. Dynamic Scheduling with Convex Delay Costs: The Generalized $c|mu$ Rule. *The Annals of Applied Probability*, 5, 1995.

[47] V. Venkataramani, Z. Amsden, N. Bronson, G. Cabrera III, P. Chakka, P. Dimov, H. Ding, J. Ferris, A. Giardullo, J. Hoon, S. Kulkarni, N. Lawrence, M. Marchukov, D. Petrov, and L. Puzar. TAO: How Facebook Serves the Social Graph. In *SIGMOD*, 2012.

[48] A. Vulimiri, P. B. Godfrey, R. Mittal, J. Sherry, S. Ratnasamy, and S. Shenker. Low Latency via Redundancy. In *CoNEXT*, 2013.

[49] Z. Wu, C. Yu, and H. V. Madhyastha. CosTLO: Cost-Effective Redundancy for Lower Latency Variance on Cloud Storage Services. In *NSDI*, 2015.

[50] Y. Xu, Z. Musgrave, B. Noble, and M. Bailey. Bobtail: Avoiding Long Tails in the Cloud. In *NSDI*, 2013.

[51] M. Zaharia, A. Konwinski, A. D. Joseph, R. Katz, and I. Stoica. Improving MapReduce Performance in Heterogeneous Environments. In *OSDI*, 2008.

[52] X. Zhang, E. Tune, R. Hagmann, R. Jnagal, V. Gokhale, and J. Wilkes. CPI^2: CPU performance isolation for shared compute clusters. In *EuroSys*, 2013.

[53] T. Zhu, A. Tumanov, M. A. Kozuch, M. Harchol-Balter, and G. R. Ganger. PriorityMeister: Tail Latency QoS for Shared Networked Storage. In *SoCC*, 2014.

CubicRing: Enabling One-Hop Failure Detection and Recovery for Distributed In-Memory Storage Systems

Yiming Zhang**, Chuanxiong Guo*†*, Dongsheng Li***, Rui Chu***, Haitao Wu*†*, Yongqiang Xiong*‡
**National University of Defense Technology,* †*Microsoft,* ‡*MSR*
{ymzhang, dsli, rchu}@nudt.edu.cn, {chguo, hwu, yqx}@microsoft.com

Abstract

In-memory storage has the benefits of low I/O latency and high I/O throughput. Fast failure recovery is crucial for large-scale in-memory storage systems, bringing network-related challenges including false detection due to transient network problems, traffic congestion during the recovery, and top-of-rack switch failures. This paper presents CubicRing, a distributed structure for cube-based networks which exploits network proximity to restrict failure detection and recovery within the smallest possible one-hop range. We leverage the CubicRing structure to address the aforementioned challenges and design a network-aware in-memory key-value store called MemCube. In a 64-node 10GbE testbed, MemCube recovers 48 GB of data for a single server failure in 3.1 seconds. The 14 recovery servers achieve 123.9 Gb/sec aggregate recovery throughput, which is 88.5% of the ideal aggregate bandwidth.

1 Introduction

Disk-based storage is becoming increasingly problematic in meeting the needs of large-scale cloud applications in terms of I/O latency, bandwidth and throughput. As a result, in recent years we see a trend of migrating data from disks to random access memory (RAM) in storage systems. In-memory storage is proposed to keep data entirely and permanently in the RAM of storage servers. E.g., Tenant's CMEM [2, 4] builds a storage cluster with thousands of servers to provide public in-memory key-value store service, which uses one *synchronous* backup server for each of the RAM storage servers. Thousands of latency-sensitive applications (e.g., online games) have stored several tens of TB of data in CMEM. Ousterhout et al. propose RAMCloud [45], an in-memory key-value store that keeps one copy of data in storage servers' RAM and stores redundant backup copies on backup servers' disks. RAMCloud uses InfiniBand networks to achieve low latency RPC.

In-memory storage has many advantages over disk-based storage including high I/O throughput, high bandwidth, and low latency. For instance, CMEM provides $1000\times$ greater throughput than disk-based systems [4]; RAMCloud boosts the performance of online data-intensive applications [46] which make a large number of sequential I/O requests in limited response time (e.g., generating dynamic HTML pages in Facebook [49]); and the applications need no longer maintain the consistency between the RAM and a separate backing store.

Fast failure recovery is crucial for large-scale in-memory storage systems to achieve high durability and availability. Previous studies [46] show for normal cases ($3\times$ replication, 2 failures/year/server with a Poisson distribution) in a 10,000-server in-memory storage system, the probability of data loss in 1 year is about 10^{-6} if the recovery is finished in 1 second; and it increases to 10^{-4} when the recovery time is 10 seconds. On the other hand, the relatively high failure rate of commodity servers requires a recovery time of no more than a few seconds to achieve continuous availability [46] in large-scale systems. According to [7], 1000+ server failures occur in one year of Google's 1800-server clusters. Since the recovery of in-memory storage server failures requires to fully utilize the resources of the cluster [45], a recovery time of a few seconds would result in an availability of about four nines (99.99%, 3,150 seconds downtime/year) if only server failures are considered, while a recovery time of several tens of seconds may degrade the availability to less than three nines, which could become the dominant factor for the overall availability.

RAMCloud realizes fast failure recovery by randomly scattering backup data on many backup servers' disks and reconstructing lost data in parallel through high-bandwidth InfiniBand networks. However, many realistic network-related challenges remain to be addressed for large-scale in-memory storage systems: (i) it is difficult to quickly distinguish transient network problems from server failures across a large-scale network; (ii) the large number (up to tens of thousands) of parallel recovery flows is likely to bring continuous traffic congestion which may result in a long recovery time; and (iii) top-of-rack (ToR) switch failures make fast failure recovery even more challenging.

To address these challenges this paper presents *CubicRing*, a distributed structure for cube-networks-based in-memory storage systems. CubicRing exploits network proximity to restrict failure detection and recovery within the smallest possible (i.e., one-hop) range, and cube networks could naturally handle switch failures with their multiple paths. We leverage the CubicRing structure to design a network-aware in-memory key-value store called *MemCube*, where the storage system and the network collaborate to achieve fast failure recovery. We implement a prototype of MemCube on BCube [35], and build a 64-node 10GbE testbed for MemCube evaluation. MemCube recovers 48 GB of data from a failed server in 3.1 seconds. In the recovery, the 14 recovery servers achieve 123.9 Gb/sec aggregate recovery throughput, which is 88.5% of the ideal aggregate bandwidth.

2 Preliminaries

Large-scale in-memory storage systems must provide a high level of durability and availability. One possible approach is to replicate all the data to the RAM of backup servers [4]. However, this approach would dramatically increase both the cost and the energy usage, and in-memory replicas are still vulnerable under power failures. On the other hand, although erasure coding can reduce some of the cost, it makes recovery considerably more expensive [46]. RAMCloud leverages high-bandwidth InfiniBand networks and utilizes aggressive data partitioning [19] for fast failure recovery [45]. It *randomly* scatters backup data across many backup servers' disks, and after a failure happens it quickly reconstructs lost data in parallel. However, it is difficult for RAMCloud's approach to scale to large clusters with thousands of servers because many network problems remain to be addressed [3]. We characterize the common network-related challenges for fast failure recovery in large-scale in-memory storage systems as follows.

False failure detection. To quickly recover from a failure, the timeout of heartbeats should be relatively short. However, various transient network problems [31] like incast and temporary hot spot may make heartbeats be discarded (in Ethernet) or suspended (in InfiniBand), making it difficult to be distinguished from real server failures. Although false failure detection is not fatal (as discussed in Section 5), the recovery of tens of GB of data is definitely very *expensive*. Since network problems cannot be completely avoided in any large-scale systems, our solution is to shorten the paths that heartbeats have to traverse, reducing the chances of encountering transient network problems. Ideally, if the servers only inspect the status of directly-connected neighbors, then we can minimize the possibility of false positives induced by transient network problems.

Recovery traffic congestion. Fast failure recovery requires an aggregate recovery bandwidth of at least tens of GB/sec both for disks and for networks. This means that hundreds or even thousands of servers will be involved in the recovery. If the distributed recovery takes place in a random and unarranged manner and the recovery flows traverse long paths, it may bring hot spots in the network and result in unexpected long recovery time. Even on full bisection bandwidth networks like FatTree [34], severe congestion is still inevitable due to the problem of ECMP (equal-cost multi-path) routing [10]: large, long-lived recovery flows may collide on their hash and end up on the same output ports creating in-network bottlenecks. To address this problem, our solution is to restrict the recovery traffic within the smallest possible range. Ideally, if all the recovery flows are one-hop, then we can eliminate the possibility of in-network congestion.

ToR switch failures. A rack usually has tens of servers connected to a ToR switch. In previous work [4, 45] when a ToR switch fails, all its servers are considered failed and several TB of data may need to be recovered. The recovery storm takes much more time than a single recovery. Since the servers connected to a failed switch are actually "alive", our solution is to build the in-memory storage system on a multi-homed cubic topology, each server being connected to multiple switches. When one switch fails, the servers can use other paths to remain connected and thus no urgent recovery is needed.

3 Structure

3.1 Design Choices

Large-scale in-memory storage systems aggregate the RAM of a large number of servers (each with at least several tens of GB of RAM) into a single storage. This subsection discusses our choices of failure model, hardware, data model, and structure.

Failure model. For storage systems, (i) servers and switches may crash, which results in data loss (omission failures) [52]; and (ii) servers and switches may experience memory/disk corruption, software bugs, etc, modifying data and sending corrupted messages to other servers (commission failures) [43]. Like RAMCloud, in this paper we mainly focus on *omission* failures. Commission failures can be detected and handled using existing techniques like Merkle-tree based, end-to-end verification and replication [43, 52], but this falls beyond the scope of this paper and is orthogonal to our design.

Network hardware. The design of CubicRing is independent to network hardware and can be applied to both Ethernet and InfiniBand. For implementation, we follow the technical trend and focus on Ethernet, because most data centers are constructed using commodity Ethernet switches and high-performance Ethernet is more promising and cost-effective [34]. Recent advances show that Ethernet switches with 100 Gbps bandwidth [6] and sub-μs latency [5] are practical in the near future, and Ethernet NICs with RDMA support have reduced much of the latency of complex protocol stacks [27].

Data model. We focus on a simple key-value store that supports arbitrary number of key-value pairs, which consist of a 64-bit key, a variable-length value, and a 64-bit version number. Our prototype provides a simple set of operations ("*set* key value", "*get* key" and "*delete* key") for writing/updating, reading and deleting data.

Primary-recovery-backup. Storage systems have multiple copies for each piece of data. There are two choices, namely *symmetric replication* [22] and *primary-backup* [16], to maintain the durability and consistency. In symmetric replication all copies have to be kept in the RAM of servers and a quorum-like technique [25] is used for conflict resolution. In contrast, in primary-backup only one primary copy is needed to be stored in RAM while redundant backup copies could be stored on disks, and all read/write operations are through the primary copy. Considering the relatively high cost and energy usage per bit of RAM, we prefer *primary-backup*.

We refer to the servers storing the primary copies in RAM as **primary servers**, and the servers storing the backup copies on disks as **backup servers**. Once a primary server fails, the backup servers will recover the backup copies to some healthy servers that are called **recovery servers**. As discussed in Section 5, the number of recovery servers is a tradeoff between recovery time and recovered data locality: a larger number decreases the recovery time but results in higher fragmentation, and vice versa. The "primary-recovery-backup" structure (shown in Fig. 1(a)) is adopted by many storage systems like RAMCloud and BigTable [19], where each server symmetrically acts as all the three roles.

3.2 CubicRing

Our basic idea is to restrict failure detection and recovery traffic within the *smallest* possible (i.e., 1-hop) range. We improve the primary-recovery-backup structure (shown in Fig. 1(a)) with a directly-connected tree (shown in Fig. 1(b)), where a primary server has multiple directly-connected recovery servers, each of which has multiple directly-connected backup servers. Here two servers are "directly-connected" if they are connected to the same switch. Clearly, Fig. 1(b) can be viewed as a special case of Fig. 1(a).

In Fig. 1(b), the primary server P periodically sends heartbeats to its recovery servers, and once the recovery servers find P failed, they will recover the lost data from their backup servers. Since the recovery servers directly connect to the primary server, they can eliminate much of the possibility of false detection due to transient network problems (as discussed in Section 4.1); and since they also directly connect to their backup servers, the recovery traffic is guaranteed to have no in-network congestion.

The directly-connected tree provides great benefit for accurate failure detection and fast recovery. We *symmetrically* map the tree onto the entire network, i.e., each server equally plays all the three roles of primary/recovery/backup server. Our insight is that all cubic topologies are some variations of generalized hypercube (GHC) [15], each vertex of which can be viewed as the root of a tree shown in Fig. 1(b).

We take BCube [35] as an example. BCube$(n,0)$ is simply n servers connected to an n-port switch. BCube$(n,1)$ is constructed from n BCube$(n,0)$ and n n-port switches. More generically, a BCube(n,k) $(k \geq 1)$ is constructed from n BCube$(n,k-1)$ and n^k n-port switches, and has $N = n^{k+1}$ servers $a_k a_{k-1} \cdots a_0$ where $a_i \in [0,n-1], i \in [0,k]$. Fig. 2 shows a BCube$(4,1)$ constructed from 4 BCube$(4,0)$. If we replace each switch and its n links of BCube(n,k) with an $n \times (n-1)$ full mesh that directly connects the servers, we will get a $(k+1)$-dimension, n-tuple generalized hypercube.

We design the multi-layer cubic rings (*CubicRing*) as shown in Fig. 3 to map the key space onto a cube-based network (e.g., BCube), following the primary-recovery-backup structure depicted in Fig. 1.

- The first layer is the *primary ring*, which is composed of all the servers. The entire key space is divided and assigned to the servers on the primary ring. Fig. 3 shows an example of the primary ring.
- Each *primary server* on the primary ring, say server P, has a second layer ring called *recovery ring* that is composed of all its 1-hop neighbors (*recovery servers*). When P fails its data will be recovered to the RAM of its recovery servers. Fig. 3 shows an example of the recovery ring (01, 02, 03, 10, 20, 30) of a primary server 00.
- Each recovery server R corresponds to a third layer ring called *backup ring*, which is composed of the *backup servers* that are 1-hop to R and 2-hop to P. The backup copies of P's data are stored on the disks of backup servers. Fig. 3 shows an example of the (six) backup rings of a primary server 00.

In the *symmetric* CubicRing depicted in Fig. 3, all the 16 primary servers have the same primary-recovery-backup structure (i.e., a directly-connected tree) with server 00. We can easily obtain the following Theorem 1, the formal proof of which is given in Appendix A.

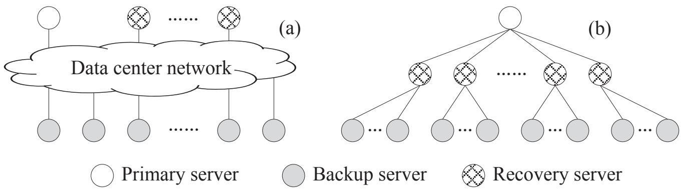


Figure 1: Primary-recovery-backup.

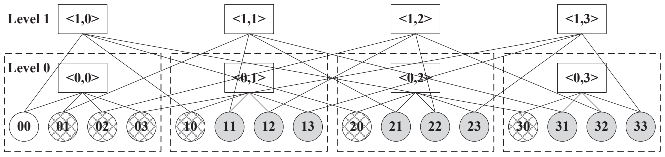


Figure 2: An example of BCube$(4,1)$.

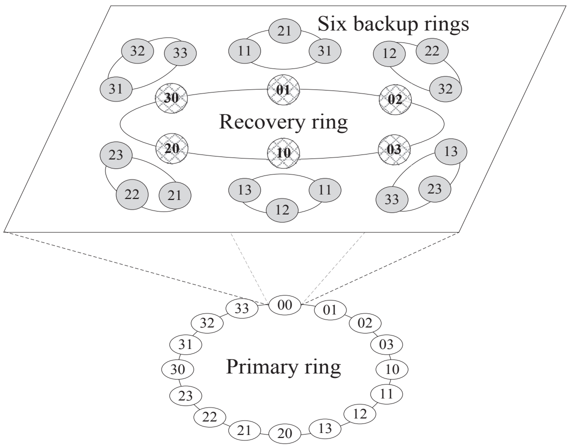


Figure 3: The CubicRing structure.

Theorem 1 *On BCube(n,k) there are n^{k+1}, $(n-1)(k+1)$, and $(n-1)k$ servers on the primary ring, recovery ring (of each primary server), and backup ring (of each recovery server), respectively. A backup server resides on two backup rings of a primary server, which has totally $\frac{(n-1)^2k(k+1)}{2}$ backup servers.*

For BCube$(16,2)$, for example, there are 4096 primary servers, each of which has 45 recovery servers (each having a 30-server backup ring) and 675 backup servers. Note that CubicRing does not require a primary server to employ all its recovery/backup servers. E.g., a primary server in BCube$(16,2)$ may employ 30 (instead of all the 45) servers on its recovery ring to reduce fragmentation, at the cost of lower aggregate recovery bandwidth.

The construction of CubicRing is applicable to all cubic topologies such as MDCube [53], *k*-ary *n*-cube [55], and hyperbanyan [29], because they are all variations of the GHC [15] topology which consists of *r*-dimensions with m_i nodes in the i^{th} dimension. A server in a particular axis is connected to all other servers in the same axis, and thus CubicRing can be naturally constructed: all the servers in a GHC form the primary ring, and for each primary server its 1-hop neighbors form its recovery ring and 2-hop neighbors form backup rings. Next, we will focus on BCube [35] to design a network-aware in-memory key-value store called *MemCube*; extending for arbitrary GHC is straightforward.

3.3 Mapping Keys to Rings

MemCube uses a *global* coordinator for managing the mapping between the key space and the primary ring. The coordinator assigns the key space to the primary servers with consideration of load balance and locality: all the primary servers should store roughly-equal size of primary copies of data (called *primary data*), and adjacent keys are preferred to be stored in one server. Currently MemCube simplifies the load balancing problem by equally dividing the key space into consecutive sub spaces, each being assigned to one primary server. This design is flexible in *dynamically* reassigning the mapping when the load distribution changes (which has not yet been implemented in our current prototype).

The key space held by a primary server *P* is further mapped to *P*'s recovery servers; and for each recovery server *R*, its sub space is mapped to its backup servers. The mapping should make all the recovery servers be assigned equal size of data, because after *P* fails they will recover *P*'s data from their backup rings simultaneously. In order to avoid potential performance bottleneck at the global coordinator, the mapping from *P*'s key space to *P*'s recovery/backup rings is maintained by *P* itself instead of the coordinator, and the recovery servers have a cache of the mapping they are involved in. After *P* fails the global coordinator asks all the recovery servers of *P* to reconstruct the mapping.

Dominant/non-dominant backup data. For durability, each primary copy has *f* backup copies, among which there are 1 *dominant* copy stored on the backup server (according to the primary-recovery-backup mapping), and $f-1$ *non-dominant* copies stored on $f-1$ secondary backup servers (*secondary servers* for short) in different *failure domains* [44]. A failure domain is a set of servers that are likely to experience a correlated failure, e.g., servers in a rack sharing a single power source. The mapping from a primary server *P*'s key space to the secondary servers is also maintained by *P* and cached at *P*'s recovery servers. Normally only the dominant backup copy participates in the recovery. Non-dominant copies are used only if the primary copy and the dominant backup copy fail concurrently. In Fig. 2, e.g., suppose that the servers connected to the same level-0 switches are in one rack failure domain. Given primary server *P* and backup server *B* of a primary copy, any $f-1$ servers that reside in $f-1$ racks different from where *P* and *B* reside can serve as secondary servers. E.g., for primary server 00 and backup server 11, the secondary servers ($f = 3$) may be 21 in rack $< 0,2 >$ and 31 in $< 0,3 >$.

Normal read/write operations. When a primary server receives a read request it directly returns. When receiving a write, it stores the new data in its RAM, and transfers it to one (2-hop-away) backup server and $f - 1$ secondary backup servers. When a backup server receives the data, it returns after storing it in its buffer to minimize write latency. The primary server returns after all the backup servers return. When the buffer fills, the backup servers use logging [48] to write buffered data to disks and free the buffer. The backup data's log is divided into *tracts* which are the unit of buffering and I/O.

4 Recovery

4.1 Failure Detection

Primary servers periodically send heartbeats to their recovery servers. If a recovery server R does not receive any heartbeats from its primary server P for a certain period, R will report this suspicious server failure to the coordinator, which would verify the problem by contacting P through all the $k+1$ paths on BCube(n,k). If P does fail, then all the tests should report the failure and the coordinator will initiate the recovery. Otherwise if some tests report P is still alive, then the coordinator will notify the available paths to the recovery servers that lose connections to P. If some (alive) servers keep being unreachable through a switch for a period of time, then the switch will be considered failed.

There are a high rate of transient network problems and a large number of small packets may be lost [31], which might result in large-scale unavailability and consequently severe disruptions. MemCube uses a relatively short timeout to achieve fast recovery. This introduces a risk that transient network problems make heartbeats get lost and thus may be incorrectly treated as server failures. Our solution is to involve as few as possible network devices/links on the paths that heartbeats traverse: MemCube achieves 1-hop failure detection which eliminates much of the possibility of network-induced false positives. In contrast, multi-hop detection (where, e.g., a heartbeat traverses $01 \rightarrow 11 \rightarrow 10 \rightarrow 00$ instead of $01 \rightarrow 00$ in Fig. 2) will considerably increase the risk.

In some uncommon cases, however, false positives are inevitable, e.g., a server is too busy to send heartbeats. MemCube uses *atomic recovery* to address this kind of problems, which will be discussed in Section 5.

4.2 Single Server Failure Recovery

A server failure means three types of failures corresponding to its three roles of primary/recovery/backup server. We sketch the major steps of recovering these failures in Pseudocode 1 (where for brevity no failure domain constraint is considered). During the recovery the backup data is read from disks of backup servers, divided into separate groups, transferred through the network, and received and processed (e.g., inserted into a hash table) by the new servers. Since most recovery flows are 1-hop, the in-network transfer is no long a bottleneck. And due to the relatively small number of recovery servers compared to other resources (as shown in our evaluation in Section 6), the recovery **bottleneck** is the *inbound* network bandwidth of recovery servers.

Pseudocode 1 Single server failure recovery

```
procedure RECOVERFAILURES(FailedServer $F$)
    Pause relevant services
    Reconstruct key space mapping of $F$
    Recover primary data for primary server failure$^*$
  ▷ All recoveries with $^*$ are performed concurrently
    Recover backup data for primary server failure$^*$
    Resume relevant services
    Recover from recovery server failure$^*$
    Recover from backup server failure
end procedure
```

Pause relevant services. After a server F's failure is confirmed by the coordinator, the key space held by F (as a primary server) will become unavailable. During the recovery all the relevant recovery/backup servers would pause servicing normal requests to avoid contention.

Reconstruct mapping. The coordinator asks all F's recovery servers to report their local cache of (part of) the mapping from F's key space to F's recovery/backup rings, and reconstructs an integrated view of the mapping previously maintained by F. Then the coordinator uses the mapping to initiate the recovery of primary/recovery/backup server failures for F.

Primary data recovery of *primary server failure*. After being notified the failure of a primary server F (say 00 in Fig. 2), F's backup servers (e.g., 11) will read backup data in tracts from disks, divide the data into separate groups for their 1-hop-away recovery servers (01 and 10), and transfer the groups of data to the recovery servers in parallel. To pipeline the processes of data transfer and storage reconstruction, as soon as the new primary server receives the first tract it begins to incorporate the new data into its in-memory storage. Keys are inserted to the hash table that maps from a key to the position where the KV resides. The new primary servers use version numbers to decide whether a key-value should be incorporated: only the highest version is retained and any older versions are discarded.

Backup data recovery of *primary server failure*. In addition to the primary data recovery, the (dominant) backup data previously stored on the old backup rings of the failed primary server F (00 in Fig. 2) needs to be recovered to the backup rings of the new primary servers R (i.e., F's recovery servers), for maintaining the CubicRing structure. To minimize the recovery time of the future failure of R (e.g., 01), the backup data of F (00) should be evenly reassigned from the old backup ring (11, 21, 31) of R (01) to the new backup rings of R. Suppose that each backup server B on the old backup ring (11, 21, 31) stores β GB of F's backup data that is previously mapped onto R (01). Since each backup server B (e.g., 11) is a new recovery server of R (01), B (11) only needs to recover its backup data to its 1-hop-away backup servers (B') on B's new backup ring (**10**, **12**, **13**), proportional to the number of recovery servers served by B': $\beta/5$ GB of backup data to 10, $2\beta/5$ GB to 12, and $2\beta/5$ GB to 13. Other old backup servers (21 and 31) have similar reassignment of their backup data, making each of the five new backup rings of R (01), namely, (**10**, **12**, **13**), (20, 22, 23), (30, 32, 33), (**12**, 22, 32) and (**13**, 23, 33), be assigned $3\beta/5$ GB of backup data from the old backup ring (11, 21, 31). For non-dominant backup data, since it does not participate in the normal recovery of primary data, MemCube does not reassign it unless the failure domain constraint is broken, as described in Section 4.3.

Resume services. After a new primary server P and P's backup servers complete the recovery of the new primary/backup data, P will update its mapping of the relevant data at its new recovery servers and the coordinator, and then P will notify the coordinator that its recovery is finished. P can choose (i) to wait for all the other new primary servers to finish their recoveries and then resume the services simultaneously (so that it will not affect others' recoveries), or (ii) to resume its service without waiting for others (so that its data can be accessed immediately). The two choices have no obvious difference in MemCube since by design all the recoveries are finished with similar time. Clients have a local cache of (part of) the mapping so that normally they can directly issue requests without querying the coordinator. If a client cannot locate a key, it fetches up-to-date information from the coordinator.

Recovery of *recovery server failure*. After a server F (e.g., 00 in Fig. 2) fails as a recovery server, for each of its primary servers P (e.g., 01), the (dominant) backup data on F's old backup ring (10, 20, 30) will be equally reassigned to the backup rings of P's remaining recovery servers R (11, 21, 31, 02, 03), in order to minimize the recovery time of P's future failure. Suppose that each backup server B on the old backup ring (10, 20, 30) stores β GB of P's (01) backup data that is previously mapped onto F (00). Then after F (00) fails, the backup data of B (10, 20, 30) will be reassigned to the 1-hop-away backup servers (B') on the backup rings of R (11, 21, 31, 02, 03), proportional to the number of recovery servers served by B': e.g., 10 will retain $\beta/5$ GB of backup data (for recovery server 11), transfer $2\beta/5$ GB to 12 (for 11 and 02), and transfer $2\beta/5$ GB to 13 (for 11 and 03). 20 and 30 have similar reassignment, making each of the five remaining backup rings be assigned $3\beta/5$ GB of backup data previously stored on 00's backup ring (10, 20, 30).

Recovery type	Size[1]	From/to[2]	Length	# flows[3]
Primary data of primary server	α	B→R	1-hop	br
Backup data of primary server	α	B→B B→R	1-hop	b^2r
Recovery server	$< \alpha$	R→B	1-hop	$(b-1)br$
Backup server	$f\alpha$	B→R	2-hop	$f(b-1)br$

[1] Total recovered size (assume a primary server stores α primary data).

[2] From the perspective of a failed *primary server*. R: recovery server. B: backup server. Bottleneck is R's inbound network bandwidth.

[3] # flows after the 1st failure. b: # backup servers on the backup ring. r: # recovery servers on the recovery ring. f: disk replication factor.

Table 1: Recovery summarization.

Recovery of *backup server failure*. After a server F (e.g., 00 in Fig. 2) fails as a backup server, its (dominant) backup data for each of its primary servers P (e.g., 11) is evenly divided and recovered from P (11) to P's 2-hop-away remaining backup servers (02, 03, 20, 30) on P's two backup rings where F (00) previously resided, to minimize the recovery time of P's future failure. Non-dominant backup data of F is recovered similarly.

Summarization. We summarize different types of recoveries in Table 1. (i) The primary/backup data recoveries of primary server failures are crucial to availability and performed concurrently. We note that there is contention between the two recoveries (B→R), but since the data size transferred to R in the backup data recovery ($S^{B\to R}_{Backup}$) is proportional to the number of new recovery servers served by R, it can be proved that $S^{B\to R}_{Backup}$ is between $\frac{1}{2b-1}$ and $\frac{1}{b}$ the size transferred to R in the primary data recovery, where b is the number of servers on the backup ring. Clearly it is negligible with relatively large b. (ii) The recovery of recovery server failures is not crucial but has no contention with primary server recovery, and thus could also be performed concurrently. (iii) The recovery of backup server failures has contention with primary server recovery (B→R), and thus should wait until the crucial recovery is finished. The deferred recovery has little affect on availability, and during this period the involved primary servers can service requests as usual, except that there is one less backup copy for the unrecovered backup data. The version numbers are used when multiple backup writes conflict (e.g., one from a new client write while another from the recovery of backup server failures).

4.3 Additional Failure Scenarios

Multiple failures. If multiple failures take place one by one, i.e., one failure happens after the previous failure has been *completely* recovered, MemCube recovers from each failure independently. Clearly the number of failures that the CubicRing structure can tolerate is bounded by the number of servers on each recovery ring: if all recovery servers of a primary server fail then the CubicRing structure fails. (If all backup servers of a recovery server R fail then it can be viewed as a failure of R.) In the worst case, e.g., the CubicRing structure on BCube$(16,2)$ can tolerate at least 44 failures, while the structure on BCube$(4,1)$ can tolerate at least 5 failures. Note that a *CubicRing* failure does NOT mean any data loss. This is because even though the data cannot be recovered to the recovery ring after the CubicRing structure fails, it still can be recovered to any healthy servers in MemCube.

Concurrent failures. When multiple failures happen concurrently, MemCube separately recovers from each failure, unless they are 1-hop or 2-hop neighbors. (i) If two directly-connected neighbors fail, e.g., during the recovery of a failed server F the coordinator cannot get responses from one of F's recovery servers, MemCube excludes them from each other's recovery ring, and recovers each failure as if it is a single failure. (ii) If a primary server and its backup server fail, the recovery server asks secondary servers for non-dominant copies.

Failure domain. MemCube guarantees after recovery none of the f backup copies are in the same domain. For instance, in the example of backup data recovery of a primary server (00) failure in Section 4.2, some dominant backup data may be reassigned to the same rack where the non-dominant data resides. E.g., the backup data previously stored on 11 is reassigned to (01, 21, 31) for the new primary server 10. In this case MemCube will reassign the affected non-dominant data to a new secondary server in a different rack.

Switch failures. In traditional storage systems a ToR switch failure results in a recovery storm, where all the abandoned servers connected to that switch are actually alive. In contrast, MemCube handles switch failures simply by leveraging the multiple paths between any two servers. Since any k switch failures in BCube(n,k) result in only graceful degradation [35] but no data loss or unavailability, it is not critical and the failed k switches can be replaced in a relatively long period of time.

5 Discussion

Over-provisioning ratio. If a primary server fails, its recovery servers must have enough RAM to accommodate the recovered data. So the RAM of all the servers need to be over-provisioned beforehand. We obtain the following Theorem 2 for the *over-provisioning ratio* (θ), the formal proof of which is given in Appendix B. In BCube$(16,2)$, e.g., if $\theta = 1.15$ then at least 6 failures can be tolerated; and if $\theta = 1.4375$ then at least 14 failures can be tolerated. In contrast, RAMCloud does not have a deterministic θ due to its randomized data placement. Note that similar to the CubicRing failure discussed in Section 4.3, if no enough RAM available on some specific servers MemCube can be simply "degraded" to RAMCloud without any data loss.

Theorem 2 *Consider a MemCube on BCube(n,k) and suppose that before any failures each server installs α GB of RAM and stores β GB of data. We define the* ***over-provisioning ratio*** *as $\theta = \alpha/\beta$. If we want to keep the CubicRing structure after the r^{th} failure in the worst case, we should have $\theta \geq 1 + \frac{r}{nk+n-k-r}$.*

Fragmentation. After a primary server fails MemCube recovers its data to multiple new servers, on which the recovered fragmented data may lost locality. Although locality has no effects on our current data model, this issue might become important if MemCube supports richer models in the future. Given a set of KVs (S), we define the *fragmentation ratio* (μ) as the initial number of primary servers responsible for S divided by the current number of servers for S. Higher μ means lower fragmentation and thus is desired for better locality. As discussed in Section 3.2, the number of recovery servers involved in a recovery is configurable. Larger numbers increase the aggregate bandwidth but result in higher fragmentation, and vice versa. We study the tradeoff between aggregate bandwidth and μ in Section 6.4. A simple method for defragmentation is to replace the failed server with a new one and restore the data.

Heterogeneity and stragglers. The backup servers may have different parameters of disk/CPU/RAM/network resources. MemCube handles heterogeneity by assigning backup data according to the bottleneck resource. E.g., if the network bandwidth of backup servers is the bottleneck for recovery, MemCube will assign backup data to them proportional to their bandwidth [45] so that they can finish the recovery with similar time.

MemCube uses a simple method to handle stragglers. Since the numbers of servers on the recovery/backup rings and the bandwidth of each server are known, MemCube can compute the expected recovered size for each server given a time window. A recovery server (R) periodically computes for each of its backup servers (B) the ratio (π_R^B) of the data size recovered from B to the expected recovered size from B within the last period. If π_R^B is lower than a pre-defined threshold, then B will be considered as a straggler and R will use B's secondary server instead. The coordinator identifies stragglers from recovery servers in a similar way, and it will reassign the straggler's responsible data to other healthy recovery servers. In both cases non-local recovery will occur, but we expect little impact on the overall performance.

Consistency guarantees. A complete discussion of consistency guarantees is beyond the scope of this work, and here we briefly discuss the main factors affecting consistency. False positives are inevitable in failure detection. Therefore, MemCube adopts *atomic recovery*, where once the coordinator declares a server P dead, it will ask all P's backup servers to (i) reject any further backup requests from P and (ii) indicate P to stop its service. Buffered backup writes from P before the declaration should be finished since they have been returned. Primary servers also periodically contact with their backup servers so that they can stop servicing pure read requests after being declared dead. Therefore, false detection in MemCube is not *fatal* (but *expensive*).

MemCube uses ZooKeeper [39] enabled coordinators to store its global mapping information between the key space and the primary servers. There are one active coordinator and several standby coordinators which are competing for a single lease, ensuring at most one coordinator to be responsible at a time. After the active coordinator fails, some standby coordinator will acquire the lease and become active. If a server fails concurrently with a coordinator failure, e.g., the recovery servers R cannot get response from the coordinator, R will ask the ZooKeeper service to locate the new active coordinator and then report the failure to it. Afterwards the normal recovery procedure is performed.

Since there are $k+1$ paths between any two servers in BCube(n,k), MemCube is unlikely to have a network partition. If this happens, an operator can stop the entire system and wait until the network recovers. Similarly, non-transitive failures [52] are unlikely since all paths to a suspiciously failed server are tested.

Operational Issues. MemCube is designed on top of BCube, which has similar cost and wiring complexity with FatTree. For isntance, both BCube and FatTree use 128 wires for building our 64-server testbed (discussed in Section 6). Clearly there might be a bandwidth waste in MemCube if the network is not busy, but the advantages of BCube include not only high bandwidth and throughput, but also fast failure detection and recovery, graceful degradation during switch failures, etc. Also we note that BCube uses COTS switches/NICs, and thus the extra cost is low and acceptable.

As described in Line 2 of Pseudocode 1 in Section 4.2, for minimizing the recovery time MemCube stops all the relevant services during the recovery. For BCube$(16,2)$ with 4096 servers, for example, given the normal failure rate (about $1 \sim 2$ failures/server/year [7]) and the recovery time (a few seconds as shown in Section 6), the "background" network utilization of recovery traffic is less than 10^{-4}.

A dangerous situation is that the entire system loses power at once. A simple way to address this problem is to install on each server a small backup battery. The battery ONLY needs to extend the power long enough to ensure that the backup server's *buffered* backup data (that is yet to be written to disks) be flushed. When power returns the cold start is performed like many concurrent recoveries of all servers.

6 Evaluation

We have implemented a prototype of MemCube by adding a *MemCube module* to memcached-1.4.15 on Linux, which contains: (i) a *connection manager* that maintains the status of neighbors and interacts with other servers; (ii) a *storage manager* that handles *set/get/delete* requests in a server's RAM and asynchronously writes backup data to disks by appending the data to its on-disk log that is divided into 8MB tracts; and (iii) a *recovery manager* that reconstructs primary/backup data (and the corresponding mapping) on the new primary/backup servers and inspects the recovery process. We also implement a simple *global coordinator* that maintains the configuration, the addresses of servers, and the mapping between the key space and the servers along with the size of data stored in each server's RAM.

6.1 Testbed

We have built a testbed with 64 PowerLeader servers and five Pronto 3780 48-port 10GbE switches. Each server has 12 Intel Xeon E5-2640 2.5GHz cores and 64 GB RAM, and installs six Hitachi 7200 RPM, 1 TB disks and one 10GbE 2-port NIC.

We use four switches to construct a 64-node BCube(8,1) network to run MemCube, where each switch acts as four 8-port virtual switches and connects to 32 servers. We also use the five switches to build a 64-node tree and a 64-node FatTree to emulate and test RAMCloud [45] on Ethernet. For tree, we simply have each of four switches connect to 16 servers and the fifth switch act as the aggregate switch, getting a relatively high over-subscription ratio of 1 : 16. For FatTree, we use three switches in the first level and two in the second. In the first level we use two switches to connect to 24 servers each and act as three 8-port virtual switches, and use the third switch to connect to 16 servers and act as two virtual switches; and each switch has the same number of ports connected to the second level switches as that to the servers. In the second level each switch acts as four 8-port virtual switches. We also build a 1 : 4 oversubscribed FatTree where the first level has

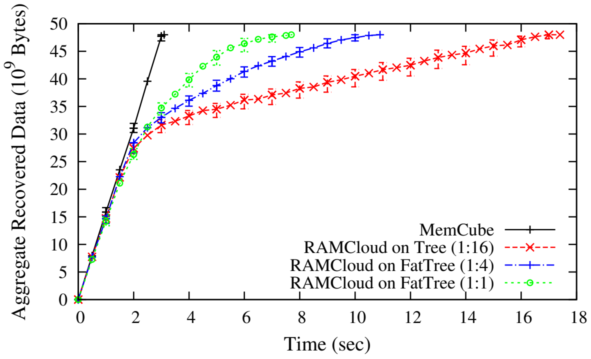


Figure 4: Single server failure recovery.

4 switches each being connected to 16 servers and the second level has the fifth switch being connected to four ports of each of the first-level switches. FatTree uses ECMP routing with hash-based path selection to achieve load balancing. Our testbed therefore supports both MemCube on BCube and RAMCloud on tree/FatTree.

All experiments use a disk replication factor of 3, i.e., 1 *primary* copy in RAM and 3 *backup* copies on servers' disks. Clearly, given the disk bandwidth of 100∼200 MB/sec and the number (6) of disks per server, the network bandwidth of 10 Gb/sec, and the ratio of recovery servers to backup servers (14/49), the bottleneck is at the inbound network bandwidth of recovery servers.

A client initially fills the 64 primary servers each with 48 GB of primary data. During the process we measure the write throughput of one MemCube primary server. We slightly modify the Redis benchmark [13, 26] to adapt to MemCube, which uses a configurable number of busy loops asynchronously writing KVs. The maximum write throughput of a single MemCube primary server is about 197.6K writes per second when it runs 8 single-threaded service processes each corresponding to 200 loops. In contrast, the maximum throughput of an *unmodified* Memcached server is about 225.5K writes per second when it runs 4 single-threaded service processes each corresponding to 250 loops.

After a failure happens, the recovery is conducted following Pseudocode 1. Our evaluation answers the following questions: How fast can MemCube recover a single server failure, even with stragglers (§6.2)? How well does MemCube perform under various patterns of failures (§6.3)? And what is the impact of using different number of recovery/backup servers (§6.4)?

6.2 Single Server Failure Recovery

We first evaluate the recovery of a single server failure in MemCube. The client sends a magic RPC to a primary server that kills its service process. The recovery procedure is started after waiting 300 milliseconds of heartbeat timeout. The coordinator waits until all the primary/backup data is recovered and reports the size of the aggregate recovered (primary) data over time. We also evaluate RAMCloud [45] both on tree and on FatTree with ECMP [34], where each primary server uses 14 recovery servers and 49 backup servers (which are the same as in MemCube).

(n,k) # servers	**(4,3)** 256	**(8,2)** 512	**(8,3)** 4096	**(16,1)** 256	**(16,2)** 4096
MemCube	1.20	1.01	0.51	1.44	0.48
RAMCloud	26.59	42.98	83.40	22.68	40.59

Table 2: Simulated recovery time (in seconds).

The result is depicted in Fig. 4. Each point is an average of 5 runs except the last points because the fast runs may have completed. MemCube recovers 48 GB of data in 3.1 seconds. The aggregate recovery throughput is about 123.9 Gb/sec, very close to the optimal aggregate bandwidth bounded by the NIC bandwidth and the number of recovery servers. Every recovery server achieves the recovery throughput of about 8.85 Gb/sec.

The recovery process of RAMCloud is also depicted in Fig. 4. On tree, RAMCloud has similar performance with MemCube in the beginning but gets a dramatic degradation after 2 seconds. This is because the recovery servers randomly choose their new backup servers without a global view of the network, and the tree has an over-subscription ratio of 1 : 16 which generates severe congestion at the root. At beginning local flows within a switch saturate the recovery servers' NICs, the aggregate bandwidth of which is the same as that in MemCube. But after the local flows complete the aggregate recovery bandwidth will drop. Non-blocking FatTree is designed to alleviate this problem, but since ECMP randomly selects paths for the flows, the full bisection bandwidth is not guaranteed but only stochastically *likely* across multiple flows. Thus long-lived recovery flows are problematic with ECMP and RAMCloud (both on tree and on FatTree) experiences long recovery time. Note that the results in Fig. 4 are worse than that in [45], where RAMCloud recovers 35 GB of data in 1.6 seconds in a 60-node cluster. This is because in [45] (i) RAMCloud uses 5 32Gbps-InfiniBand switches to build the testbed (while we emulate 16 8-port 10GbE switches); and (ii) it uses all the nodes as recovery servers for the failed server (while we use only 14 recovery servers).

We also evaluate the recovery for larger scales of MemCube (on BCube(n,k)) and RAMCloud (on non-blocking FatTree) through simulations. Since in most cases the bottleneck is at the bandwidth of recovery servers, we simplify the simulations by using NS2 [8] to simulate the process of transferring primary/backup data for the failed server which has 48 GB of primary data. The result is summarized in Table 2, where the first

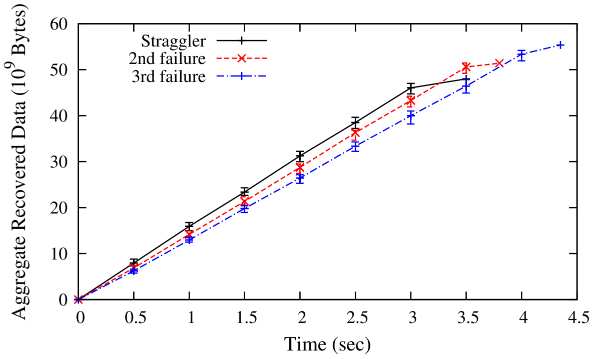


Figure 5: Straggler and multiple failures.

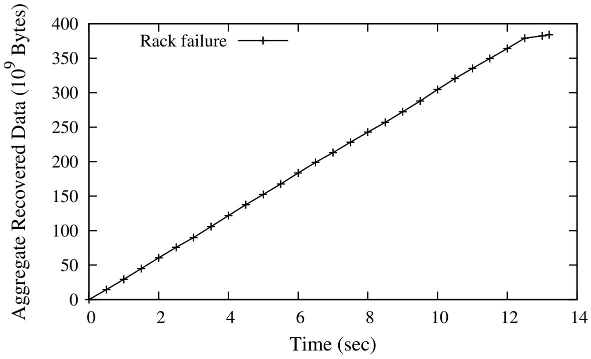


Figure 6: Rack failure recovery.

row lists different combinations of (n,k) and the numbers of servers, and the next two rows respectively show the corresponding recovery time (in seconds) for MemCube and RAMCloud. Note that we only simulate the process of primary/backup data transfer and ignore the failure detection time (a few hundred milliseconds), the coordination time (100+ milliseconds), the time of reading the first tract from disks (about 100 milliseconds), and the potential bottleneck at CPU which is because a recovery server uses k NIC ports for recovery. Therefore, although the simulated recovery time for both BCube(8,3) and BCube(16,2) is only about half a second, in practice it would be difficult to recover faster than 1 second.

We emulate a straggler during the recovery by limiting a backup server's outbound bandwidth to $1/3$ the bandwidth in normal recovery ($\frac{123.9}{49\times3} \approx 0.84$ Gb/sec). In this experiment, every new primary server R computes π_R^B (defined in Section 5) for each of its backup server B every half a second, the threshold is set to 0.7, and the straggler occurs 1 second after the recovery begins. The recovery procedure is depicted in Fig. 5 (denoted as *Straggler*), each point of which is an average of 5 runs. The result shows that MemCube performs well after the straggler occurs by using other backup flows to saturate the recovery server's spare bandwidth. After 1.5 seconds MemCube will detect the straggler and use its secondary server instead, which finishes its recovery at about 3.5 seconds. The additional time compared to MemCube's normal recovery is because the straggler recovers less data than others between 1 and 1.5 seconds.

6.3 Multiple Server Failures Recovery

We evaluate the recovery of multiple failures with the one-by-one pattern, i.e., one failure happens after the previous failure has already been completely recovered. All subsequent failures happen on the same recovery ring of the first failure, which generates the worst-case scenario. The result is depicted in Fig. 5, which shows the size of recovered data over time for the second and third failures. Each point is an average of 5 runs. This figure shows that there is a graceful degradation of recovery performance as failures happen one by one, mainly due to the decrease of the number of recovery servers and the increase of the primary data size.

We then emulate a rack failure by sending a magic RPC to the 8 servers connected to an 8-port virtual switch to kill their MemCube processes. Currently our prototype only supports rack failure recovery of primary/backup data for primary servers failures (Lines 4 and 5 in Pseudocode 1), but extending for supporting the other two types of recovery (Lines 7 and 8) is straightforward. The recovery procedure is depicted in Fig. 6, where each point is an average of 5 runs and the differences to the mean are less than 5% (omitted here for clarity). MemCube recovers a rack failure of 8 primary servers in about 13.2 seconds, Compared with the single failure recovery, the recovered data size increases by 8×, the total number of recovery servers increases by 4×, and the recovery time increases by about 4×, meaning the per-server recovery throughput is only about $1/2$ that in single failure recovery. This is because both primary data and backup data are recovered from *all* servers to *all* servers, in contrast in single failure recovery (as discussed in Section 4.2) only $\frac{1}{2b-1} = 1/13$ of the backup data recovery contends with the primary data recovery, where $b = 7$ is the number of servers on a backup ring. Clearly, even when multiple primary-recovery-backup structures overlap there is still no severe competition during the recoveries of multiple concurrent failures.

MemCube handles a switch failure with graceful performance degradation by leveraging the multiple paths of BCube. To evaluate this, we first measure the write throughput of a primary server, disable a switch connected to that server, wait for 1 second, and then measure the write throughput again. Running 8 single-threaded server processes, before the switch failure the write throughput $\approx$ 197.6K writes/sec. After the switch failure the write throughput $\approx$ 162.2K writes/sec with a degradation of less than 18%, which is mainly because the redundant paths traverse more intermediate nodes.

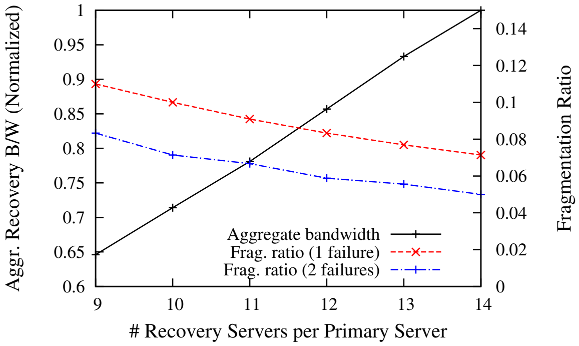


Figure 7: Tradeoff of different # recovery servers.

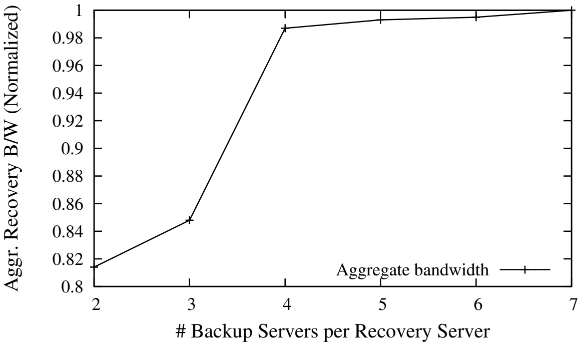


Figure 8: Impact of different # backup servers.

6.4 Impact of # Recovery/Backup Servers

We study the impact of different number of recovery servers used by a primary server on the average aggregate recovery bandwidth and the *fragmentation ratio* (introduced in Section 5). The bandwidth is computed as the recovered size (of primary data) divided by the recovery time and normalized to the baseline with all the 14 recovery servers being used. Each recovery server adopts all its 7 backup servers. As shown in Fig. 7, with the increase of the number of recovery servers, MemCube gets an almost linear speedup for the bandwidth, while the ratio decreases (meaning more severe fragmentation) after the recovery of both 1 and 2 failures. In practice MemCube can choose the tradeoff between fragmentation and recovery bandwidth by using different number of recovery servers accordingly.

We study the impact of different number of backup servers (b) used by a recovery server on the aggregate recovery bandwidth. The primary server uses all its 14 recovery servers. Since when b is small the bandwidth of backup servers (BW_B) may become the bottleneck instead of the bandwidth of recovery servers (BW_R), the primary data is assigned to the recovery servers R according to R's $\min(BW_R, \sum_{B\in Ring(R)} BW_B)$. As shown in Fig. 8, when the number of backup servers is small ($b = 2,3$) the bandwidth of backup servers is the bottleneck. This is because the data sent by backup servers is twice as much as that received by recovery servers due to the concurrent recovery of backup data (and note that 1 backup server serves 2 recovery servers). When b increases to 4, the aggregate bandwidth is almost the same as the baseline ($b = 7$), because since then the performance is again bounded by the bandwidth of recovery servers. Although the number of backup servers has little impact on the recovery time when $b \geq 4$, we suggest to use all the backup servers to (i) achieve high CubicRing durability, and (ii) prevent the aggregate disk bandwidth to become a bottleneck when the number of disks per server is smaller than our configuration ($= 6$).

7 Related Work

Efficient KV. The idea of permanently storing data in RAM is not new. E.g., in-memory databases [30] and transaction processing systems [40] keep entire databases in the RAM of one or several servers and support full RDBMS semantics. RAMCloud [45] is proposed as a large-scale in-memory key-value store where the data is kept entirely in RAM and the backup copies are scattered across many servers' disks. RAMCloud utilizes high-bandwidth InfiniBand networks to achieve fast failure recovery. MemCube inherits some key designs of RAMCloud including the primary-recovery-backup architecture and the coordinator for key space management. MemCube improves RAMCloud by leveraging CubicRing to address several critical network-related issues, including false failure detection due to transient problems, recovery traffic congestion, and ToR switch failures. Besides, MemCube is implemented on Ethernet which has cost and scalability benefits.

Redis [13] is a key-value store that keeps all data in RAM. Redis has a richer data model than MemCube, e.g., atomic increment and transactions. However, it can prevent data loss ONLY when it is used in the *flushing mode*, where every write has to be logged to disks before it returns. MemC3 [28] improves Memcached [9] by incorporating the CLOCK replacement algorithm [1] and Concurrent Cuckoo hashing [47]. It serves up to 3× as many queries per second. MemCube can easily migrate from Memcached to MemC3 for higher throughput.

Flash memory is receiving increasing attention for flash-based storage systems (e.g., SILT [42], FAWN [12], FlashStore [23], SkimpyStash [24], and HashCache [11]). One disadvantage of in-memory storage systems is the high cost and energy usage per bit. However, when considering cost per operation, RAM is about 1000× more efficient than disk and 10× than flash memory [46]. Andersen et al. [12] and Ousterhout et al. [46] separately generalize Jim Gray's rule [33] and conclude that (i) for high access rates and small data sizes RAM is the cheapest; and (ii) the applicability of in-memory storage will be continuously increasing.

Detection/recovery. Failure detection has been widely studied in the context of monitoring remote elements by using end-to-end timeouts [18, 14, 20, 50, 37]. E.g., the ϕ-accrual detector [37] provides a measurement of detection confidence and lets applications decide corresponding actions. Recently, Falcon [41] and Pigeon [36] propose to install sensors to obtain low-level information of hardware, OS, processes, and routers/switches, to aid diagnosis. These works are complementary to MemCube and can help to provide more accurate detection. E.g., MemCube could use Pigeon to check backup servers' SMART [51] data to pre-warn disk problems. MemCube's local detection eliminates the necessity of installing code in switches (which makes Pigeon less applicable in real deployment) for congestion detection. Host failure recovery techniques (e.g., microreboot [21, 17]) focus on masking and containing failures, which can be directly applied to MemCube: a MemCube recovery will not take place until failures cannot be masked.

FDS [44] is a locality-oblivious blob store. It recovers 92 GB of data in parallel in 6.2 seconds on a 256-server 10GbE FatTree, which is less efficient compared with MemCube. Thus although FDS claims "locality is unnecessary", we show locality does matter in fast failure recovery. Similar to FDS, D-Streams [54] use parallel recovery for reliable distributed stream processing.

8 Conclusion

This paper's top-level contribution is architectural: We suggest to exploit network proximity in distributed systems to restrict failure detection and recovery within the smallest possible range, in order to minimize the uncertainty and contention induced by the network. We apply this principle to fast failure recovery of an in-memory key-value store (MemCube) by constructing the CubicRing structure: All the servers form a primary ring, and for each primary server its 1-hop neighbors form a recovery ring and 2-hop neighbors form backup rings. As failures happen, MemCube (i) leverages the CubicRing structure to quickly recover lost data, and (ii) maintains the structure.

We plan to improve MemCube in several aspects including rich data model, indices, efficient log cleaning/optimizing, super columns [19], strong consistency (i.e., linearizability [38, 32]) guarantees, and low-latency serializable transactions [56]. We also plan to implement automatic reassignment of the key space mapping when the load distribution dynamically changes. On the other hand, MemCube depends on cubic topologies, and how to apply the proposed principle to tree-based networks (e.g., FatTree) is still an open issue.

Acknowledgement

This work was supported by the National Basic Research Program of China (973) under Grant No. 2011CB302601, and the National Natural Science Foundation of China (NSFC) under Grant No. 61379055 and 61222205. We thank Ji Wang, Huiba Li, Zhihui Sun, Rodrigo Fonseca, the Pronto tech support team, and the anonymous reviewers for their help to improve this paper. This work was performed when the first author was visiting MSRA, and we thank Guohan Lu and Jiaxin Cao for the insightful comments. Some parts of the paper were written when the first author was visiting the NetOS group of Computer Lab at University of Cambridge, and we thank Professor Jon Crowcroft for his accommodation and Anil Madhavapeddy, Ripduman Sohan and Hao Liu for the discussion. Some preliminary results of this work were presented at HotCloud'12.

Appendix

A. Proof of Theorem 1

BCube(n,k) is equal to an n-tuple, $k+1$ dimensional generalized hypercube and there are n^{k+1} servers on the primary ring. Each primary server connects to $k+1$ switches, each with $n-1$ recovery servers. So there are $(n-1)(k+1)$ servers on the recovery ring. Each recovery server connects to k switches (except the one it uses to connect to its primary server), each with $n-1$ backup servers. So there are $(n-1)k$ servers on the backup ring. The backup servers is two hops away from their primary server, and thus they have exactly two digits different from their primary server. Thus each backup server services 2 recovery servers irrespective of n and k. Therefore each primary server has totally $(n-1)(k+1)\times(n-1)k/2 = \frac{(n-1)^2k(k+1)}{2}$ backup servers.

B. Proof of Theorem 2

By Theorem 1, at the beginning there are $(n-1)(k+1)$ servers on the recovery ring. Let $m=(n-1)(k+1)$. After the first server fails, MemCube must satisfy $\beta+\frac{\beta}{m}=\beta(1+\frac{1}{m})\leq\alpha$; after the second server fails, which in the worst case may be a recovery server of the first failed server, MemCube should satisfy $\beta+\frac{\beta}{m}+\frac{1}{m-1}(\beta+\frac{\beta}{m})<\beta(1+\frac{1}{m-1})^2\approx\beta(1+\frac{2}{m-1})\leq\alpha$; $\ldots$; and by parity of reasoning, after the r^{th} failure (reasonably assuming $m-r>>1$), MemCube should satisfy $\beta(1+\frac{r}{m-r+1})\leq\alpha$. Therefore, if we want to keep the CubicRing structure after the r^{th} failure in the worst case (where a subsequent failure always happens on a server that is a recovery server in the previous failure recovery), the over-provisioning ratio θ should satisfy $\theta=\alpha/\beta\geq1+\frac{r}{m+1-r}=1+\frac{r}{nk+n-k-r}$.

References

[1] http://books.google.com/books?id=5wDQNwAACAAJ.

[2] http://kylinx.com/papers/cmem1.4.pdf.

[3] https://ramcloud.stanford.edu/wiki/pages/viewpage.action?pageId=8355860sosp-2011-reviews-and-comments-on-ramcloud.

[4] http://wiki.open.qq.com/wiki/CMEM.

[5] http://www.aristanetworks.com/en/products/7100series.

[6] http://www.broadcom.com/press/release.php?id=s634491.

[7] http://www.datacenterknowledge.com/archives/2008/05/30/failure-rates-in-google-data-centers/.

[8] http://www.isi.edu/nsnam/ns/.

[9] http://www.memcached.org/.

[10] Al-Fares, M., Radhakrishnan, S., Raghavan, B., Huang, N., and Vahdat, A. Hedera: Dynamic flow scheduling for data center networks. In *NSDI* (2010), pp. 281–296.

[11] Anand, A., Muthukrishnan, C., Kappes, S., Akella, A., and Nath, S. Cheap and large cams for high performance data-intensive networked systems. In *NSDI* (2010), USENIX Association, pp. 433–448.

[12] Andersen, D. G., Franklin, J., Kaminsky, M., Phanishayee, A., Tan, L., and Vasudevan, V. Fawn: a fast array of wimpy nodes. In *SOSP* (2009), J. N. Matthews and T. E. Anderson, Eds., ACM, pp. 1–14.

[13] Antirez. An update on the memcached/redis benchmark. http://antirez.com/post/update-on-memcached-redis -benchmark.html.

[14] Bertier, M., Marin, O., and Sens, P. Implementation and performance evaluation of an adaptable failure detector. In *2002 International Conference on Dependable Systems and Networks (DSN 2002), 23-26 June 2002, Bethesda, MD, USA, Proceedings* (2002), pp. 354–363.

[15] Bhuyan, L. N., and Agrawal, D. P. Generalized hypercube and hyperbus structures for a computer network. *IEEE Trans. Computers 33*, 4 (1984), 323–333.

[16] Budhiraja, N., Marzullo, K., Schneider, F. B., and Toueg, S. The primary-backup approach, 1993.

[17] Candea, G., Kawamoto, S., Fujiki, Y., Friedman, G., and Fox, A. Microreboot - A technique for cheap recovery. In *6th Symposium on Operating System Design and Implementation (OSDI 2004), San Francisco, California, USA, December 6-8, 2004* (2004), pp. 31–44.

[18] Chandra, T. D., and Toueg, S. Unreliable failure detectors for reliable distributed systems. *J. ACM 43*, 2 (1996), 225–267.

[19] Chang, F., Dean, J., Ghemawat, S., Hsieh, W. C., Wallach, D. A., Burrows, M., Chandra, T., Fikes, A., and Gruber, R. Bigtable: A distributed storage system for structured data. In *OSDI* (2006), pp. 205–218.

[20] Chen, W., Toueg, S., and Aguilera, M. K. On the quality of service of failure detectors. *IEEE Trans. Computers 51*, 5 (2002), 561–580.

[21] Cully, B., Lefebvre, G., Meyer, D. T., Feeley, M., Hutchinson, N. C., and Warfield, A. Remus: High availability via asynchronous virtual machine replication. (best paper). In *5th USENIX Symposium on Networked Systems Design & Implementation, NSDI 2008, April 16-18, 2008, San Francisco, CA, USA, Proceedings* (2008), p. 161.

[22] Daniels, D., Doo, L. B., Downing, A., Elsbernd, C., Hallmark, G., Jain, S., Jenkins, B., Lim, P., Smith, G., Souder, B., and Stamos, J. Oracle's symmetric replication technology and implications for application design. In *SIGMOD* (1994), ACM Press, p. 467.

[23] Debnath, B. K., Sengupta, S., and Li, J. Flashstore: High throughput persistent key-value store. *PVLDB 3*, 2 (2010), 1414–1425.

[24] Debnath, B. K., Sengupta, S., and Li, J. Skimpystash: Ram space skimpy key-value store on flash-based storage. In *SIGMOD Conference* (2011), T. K. Sellis, R. J. Miller, A. Kementsietsidis, and Y. Velegrakis, Eds., ACM, pp. 25–36.

[25] DeCandia, G., Hastorun, D., Jampani, M., Kakulapati, G., Lakshman, A., Pilchin, A., Sivasubramanian, S., Vosshall, P., and Vogels, W. Dynamo: amazon's highly available key-value store. In *SOSP* (2007), pp. 205–220.

[26] Dormando. Redis vs memcached (slightly better bench). http://dormando.livejournal.com/525147.html.

[27] Dragojević, A., Narayanan, D., Castro, M., and Hodson, O. Farm: Fast remote memory. In *11th USENIX Symposium on Networked Systems Design and Implementation (NSDI 14)* (Seattle, WA, 2014), USENIX Association, pp. 401–414.

[28] Fan, B., Andersen, D. G., and Kaminsky, M. Memc3: Compact and concurrent memcache with dumber caching and smarter hashing. In *Proceedings of the 10th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2013, Lombard, IL, USA, April 2-5, 2013* (2013), pp. 371–384.

[29] Ferner, C. S., and Lee, K. Y. Hyperbanyan networks: A new class of networks for distributed memory multiprocessors. *IEEE Transactions on Computers 41*, 3 (1992), 254–261.

[30] Garcia-Molina, H., and Salem, K. Main memory database systems: An overview. *IEEE Trans. Knowl. Data Eng. 4*, 6 (1992), 509–516.

[31] Gill, P., Jain, N., and Nagappan, N. Understanding network failures in data centers: measurement, analysis, and implications. In *SIGCOMM* (2011), pp. 350–361.

[32] Glendenning, L., Beschastnikh, I., Krishnamurthy, A., and Anderson, T. E. Scalable consistency in scatter. In *SOSP* (2011), pp. 15–28.

[33] Gray, J., and Putzolu, G. R. The 5 minute rule for trading memory for disk accesses and the 10 byte rule for trading memory for cpu time. In *Proceedings of the Association for Computing Machinery Special Interest Group on Management of Data 1987 Annual Conference, San Francisco, California, May 27-29, 1987* (1987), U. Dayal and I. L. Traiger, Eds., ACM Press, pp. 395–398.

[34] Greenberg, A. G., Hamilton, J. R., Jain, N., Kandula, S., Kim, C., Lahiri, P., Maltz, D. A., Patel, P., and Sengupta, S. Vl2: a scalable and flexible data center network. *Commun. ACM 54*, 3 (2011), 95–104.

[35] 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 *SIGCOMM* (2009), pp. 63–74.

[36] Gupta, T., Leners, J. B., Aguilera, M. K., and Walfish, M. Improving availability in distributed systems with failure informers. In *Proceedings of the 10th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2013, Lombard, IL, USA, April 2-5, 2013* (2013), pp. 427–441.

[37] Hayashibara, N., Défago, X., Yared, R., and Katayama, T. The Φ accrual failure detector. In *23rd International Symposium on Reliable Distributed Systems (SRDS 2004), 18-20 October 2004, Florianpolis, Brazil* (2004), pp. 66–78.

[38] HERLIHY, M., AND WING, J. M. Linearizability: A correctness condition for concurrent objects. *ACM Trans. Program. Lang. Syst. 12*, 3 (1990), 463–492.

[39] HUNT, P., KONAR, M., JUNQUEIRA, F. P., AND REED, B. Zookeeper: Wait-free coordination for internet-scale systems. In *USENIX ATC* (2010), pp. 1–14.

[40] KALLMAN, R., KIMURA, H., NATKINS, J., PAVLO, A., RASIN, A., ZDONIK, S. B., JONES, E. P. C., MADDEN, S., STONEBRAKER, M., ZHANG, Y., HUGG, J., AND ABADI, D. J. H-store: a high-performance, distributed main memory transaction processing system. *PVLDB 1*, 2 (2008), 1496–1499.

[41] LENERS, J. B., WU, H., HUNG, W., AGUILERA, M. K., AND WALFISH, M. Detecting failures in distributed systems with the falcon spy network. In *Proceedings of the 23rd ACM Symposium on Operating Systems Principles 2011, SOSP 2011, Cascais, Portugal, October 23-26, 2011* (2011), pp. 279–294.

[42] LIM, H., FAN, B., ANDERSEN, D. G., AND KAMINSKY, M. Silt: a memory-efficient, high-performance key-value store. In *SOSP* (2011), pp. 1–13.

[43] MAHAJAN, P., SETTY, S. T. V., LEE, S., CLEMENT, A., ALVISI, L., DAHLIN, M., AND WALFISH, M. Depot: Cloud storage with minimal trust. In *9th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2010, October 4-6, 2010, Vancouver, BC, Canada, Proceedings* (2010), pp. 307–322.

[44] NIGHTINGALE, E. B., ELSON, J., FAN, J., HOFMANN, O., HOWELL, J., , AND SUZUE, Y. Flat datacenter storage. In *OSDI* (2012).

[45] ONGARO, D., RUMBLE, S. M., STUTSMAN, R., OUSTERHOUT, J. K., AND ROSENBLUM, M. Fast crash recovery in ramcloud. In *SOSP* (2011), pp. 29–41.

[46] OUSTERHOUT, J. K., AGRAWAL, P., ERICKSON, D., KOZYRAKIS, C., LEVERICH, J., MAZIÈRES, D., MITRA, S., NARAYANAN, A., PARULKAR, G. M., ROSENBLUM, M., RUMBLE, S. M., STRATMANN, E., AND STUTSMAN, R. The case for ramclouds: scalable high-performance storage entirely in dram. *Operating Systems Review 43*, 4 (2009), 92–105.

[47] PAGH, R., AND RODLER, F. F. Cuckoo hashing. *J. Algorithms 51*, 2 (2004), 122–144.

[48] ROSENBLUM, M., AND OUSTERHOUT, J. K. The design and implementation of a log-structured file system. In *SOSP* (1991), pp. 1–15.

[49] RUMBLE, S. M., ONGARO, D., STUTSMAN, R., ROSENBLUM, M., AND OUSTERHOUT, J. K. It's time for low latency. In *HotOS* (2011).

[50] SO, K. C. W., AND SIRER, E. G. Latency and bandwidth-minimizing failure detectors. In *Proceedings of the 2007 EuroSys Conference, Lisbon, Portugal, March 21-23, 2007* (2007), pp. 89–99.

[51] STEVENS, C. E. At attachment 8 - ata/atapi command set. *Technical Report 1699, Technical Committee T13* (2008).

[52] WANG, Y., KAPRITSOS, M., REN, Z., MAHAJAN, P., KIRUBANANDAM, J., ALVISI, L., AND DAHLIN, M. Robustness in the salus scalable block store. In *Proceedings of the 10th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2013, Lombard, IL, USA, April 2-5, 2013* (2013), pp. 357–370.

[53] WU, H., LU, G., LI, D., GUO, C., AND ZHANG, Y. Mdcube: a high performance network structure for modular data center interconnection. In *CoNEXT* (2009), J. Liebeherr, G. Ventre, E. W. Biersack, and S. Keshav, Eds., ACM, pp. 25–36.

[54] ZAHARIA, M., DAS, T., LI, H., HUNTER, T., SHENKER, S., AND STOICA, I. Discretized streams: Fault-tolerant streaming computation at scale. In *SOSP* (2013).

[55] ZHANG, Y., AND LIU, L. Distributed line graphs: A universal technique for designing dhts based on arbitrary regular graphs. *IEEE Trans. Knowl. Data Eng. 24*, 9 (2012), 1556–1569.

[56] ZHANG, Y., POWER, R., ZHOU, S., SOVRAN, Y., AGUILERA, M. K., AND LI, J. Transaction chains: achieving serializability with low latency in geo-distributed storage systems. In *SOSP* (2013), pp. 276–291.

CosTLO: Cost-Effective Redundancy for Lower Latency Variance on Cloud Storage Services

Zhe Wu[*‡], Curtis Yu[*], and Harsha V. Madhyastha[*‡]
[*]UC Riverside [‡]University of Michigan

Abstract—**We present CosTLO, a system that reduces the high latency variance associated with cloud storage services by augmenting GET/PUT requests issued by end-hosts with redundant requests, so that the earliest response can be considered. To reduce the cost overhead imposed by redundancy, unlike prior efforts that have used this approach, CosTLO *combines* the use of multiple forms of redundancy. Since this results in a large number of configurations in which CosTLO can issue redundant requests, we conduct a comprehensive measurement study on S3 and Azure to identify the configurations that are viable in practice. Informed by this study, we design CosTLO to satisfy any application's goals for latency variance by 1) estimating the latency variance offered by any particular configuration, 2) efficiently searching through the configuration space to select a cost-effective configuration among the ones that can offer the desired latency variance, and 3) preserving data consistency despite CosTLO's use of redundant requests. We show that, for the median PlanetLab node, CosTLO can halve the latency variance associated with fetching content from Amazon S3, with only a 25% increase in cost.**

1 Introduction

Minimizing user-perceived latencies is critical for many applications as even hundreds of milliseconds of additional delay can significantly lower revenue [19, 10, 35]. Large-scale cloud services aid application providers in this regard by enabling them to serve every user from the closest among several geographically distributed data centers. For example, our measurements from over 120 PlanetLab nodes across the globe show that, when every node downloads 1 KB-sized objects from the closest Microsoft Azure data center, the median download latency is less than 100ms for over 90% of the nodes.

However, on today's cloud services, both fetching and storing content are associated with high latency variance. For example, for over 70% of the same 120 nodes considered above, the 99^{th} percentile and median download latencies from the closest Azure data center differ by 100ms or more. These high tail latencies are problematic both for popular applications where even 1% of traffic corresponds to a significant volume of requests [23], and for applications where a single request issued by an end-host requires the application to fetch several objects (e.g., web page loads) and user-perceived latency is constrained by the object fetched last. For example, our measurements show that latency variance in S3 more than doubles the *median* page load time for 50% of PlanetLab nodes when fetching a webpage containing 50 objects.

To enable application providers to avail of the cost benefits enabled by cloud services, without having latency variance degrade user experience, we develop CosTLO (Cost-effective Tail Latency Optimizer). Since we observe that the high latency variance is caused predominantly by isolated latency spikes, CosTLO uses the well-known approach [38, 22] for reducing variance by augmenting every GET/PUT request with a set of redundant requests, so that the earliest response can be considered. We tackle three key challenges in using this redundancy-based approach in CosTLO.

First, the end-to-end latency when any end-host uploads to or downloads from a cloud storage service has several components: latency over the Internet, latency over the cloud service's data center network, and latency within the storage service. To tackle the variance in all of these components, CosTLO exploits the fact that redundant requests to cloud storage services can be issued in a variety of ways, each of which impacts a different component of end-to-end latency. For example, while issuing redundant requests to *the same object* may elicit an earlier response due to differences in load across servers hosting replicas of the object, one can further reduce the impact of server load by issuing redundant requests to *a set of objects* which are all copies of the object being accessed. Alternatively, to reduce the impact of spikes in data center network latency, redundant requests can be issued to *different front-ends* of the storage service or relayed to the *same front-end via different virtual machines (VMs)*. Furthermore, when a client is accessing an object stored in a particular data center, redundant requests can be issued to *copies of the object in other data centers* in order to tackle the variance in Internet latencies.

However, not all forms of redundancy have utility in practice due to the complex architectures of cloud services. Therefore, second, we empirically evaluate the ways in which redundant requests should be issued for CosTLO's approach to be viable on Amazon S3 and Microsoft Azure, the two largest cloud storage services today. For example, when issuing concurrent requests to multiple data centers, we find that it is essential to leverage storage services offered by multiple cloud providers; utilizing a single cloud provider's data centers is insufficient to tame the variance in Internet latencies. Our study also shows that, due to load balancing within the data center networks of cloud services, concurrent requests to the same front-end of a storage service are sufficient to tackle spikes in data center network latencies, and more complex approaches are unnecessary. To the best of our knowledge, this is the first work that identifies the key causes for latency variance in cloud storage services and studies the impact of different forms of redundancy.

Third, the number of configurations in which CosTLO can implement redundancy is unbounded—not only can CosTLO *combine* the use of various forms of redundancy, but it can also *vary* the number of redundant requests, the probability with which it issues redundant requests, etc.—and the impact on cost and latencies varies significantly across configurations. Therefore, for CosTLO to add redundancy in a manner that satisfies an application's goals for latency variance cost-effectively, it becomes essential that CosTLO be able to 1) *estimate*, rather than measure, the cost and latencies associated with any particular configuration, and 2) *search* for a cost-effective configuration, instead of enumerating through all possible configurations. To address these challenges, 1) we model the load balancing and replication within cloud storage services in order to accurately capture the dependencies between concurrent requests, and 2) we develop an efficient algorithm to identify a cost-effective CosTLO configuration that can keep latency variance below a target. Note that no prior work that uses redundant requests seeks to minimize cost.

We have implemented and deployed CosTLO across all data centers in S3 and Azure. To evaluate CosTLO, we use PlanetLab nodes at 120 sites as clients and replay a trace of Wikipedia's workload. Our results show that CosTLO can reduce the spread between 99^{th} percentile and median GET latencies by 50% for the median PlanetLab node, with only a 25% increase in cost.

2 Characterizing Latency Variance

We begin with a measurement study of Amazon S3 and Microsoft Azure. We 1) quantify the latency variance when using these services, 2) analyze the impact of latency variance on applications, and 3) identify the dominant causes of this variance.

Overview of measurements. To analyze client-perceived latencies when downloading from and uploading to cloud storage services, we gather two types of measurements for a week. First, we use 120 PlanetLab nodes across the world as representative end-hosts. Once every 3 seconds, every node uploaded a new object to and downloaded a previously stored object from the S3 and Azure data centers to which the node has the lowest median RTT. Second, from "small instance" VMs in every S3 and every Azure data center, we issued one GET and one PUT per second to the local storage service. In all cases, every GET from a data center was for a 1 KB object selected at random from 1M objects of that size stored at that data center, and every PUT was for a new 1 KB object. To minimize the impact of client-side overheads, we measure GET and PUT latencies on PlanetLab nodes as well as on VMs using timings from tcpdump.

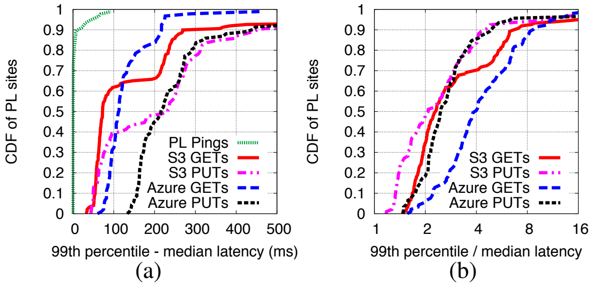


Figure 1: *(a) Absolute and (b) relative inflation in* 99^{th} *percentile latency with respect to median. Logscale x-axis in (b).*

In addition, we leverage logs exported by S3 [7] and Azure [9] to break down end-to-end latency minus DNS resolution time into its two components: 1) latency within the storage service (i.e., duration between when a request was received at one of the storage service's front-ends and when the response left the storage service), and 2) latency over the network (i.e., for the request to travel from the end-host/VM to a front-end of the storage service and for the response to travel back). We extract storage service latency directly from the storage service logs, and we can infer network latency by subtracting storage service latency from end-to-end request latency.

Quantifying latency variance. Figure 1 shows the distribution across nodes of the spread in latencies; for every node, we plot the absolute and relative difference between the 99^{th} percentile and median latencies. In both Azure and S3, the median PlanetLab node sees an absolute inflation greater than 200ms (70ms) in the 99^{th} percentile PUT (GET) latency as compared to the median latency; the median relative inflation is greater than 2x in both PUTs and GETs. To show that this high latency variance is not due to high load or slow access links of PlanetLab nodes, Figure 1 also plots for every node the difference between 99^{th} percentile and median latency to the node closest to it among all PlanetLab nodes.

Impact on applications. To show that high latency variance can significantly degrade application performance, we conduct measurement studies in two application scenarios. The first one is a webservice that serves static webpages containing 50 objects. The second one is a social network application, where an update from a user triggers a synchronization mechanism to make all of the user's followers fetch that update. In both applications, one user-level request requires the application to issue several requests to cloud storage, and user-perceived latency is constrained by the request that finishes last. We consider a setting in which (1) users only fetch objects from their closest data centers, (2) every user in the social network application has 200 followers [1], and (3) users and their followers have the same closest data centers.

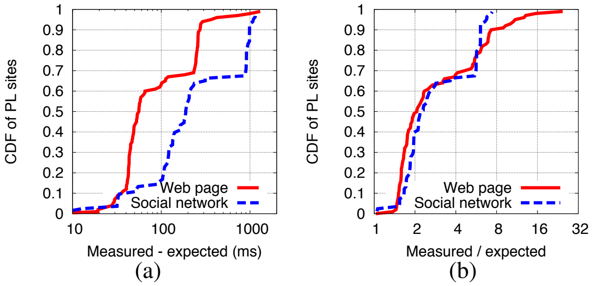
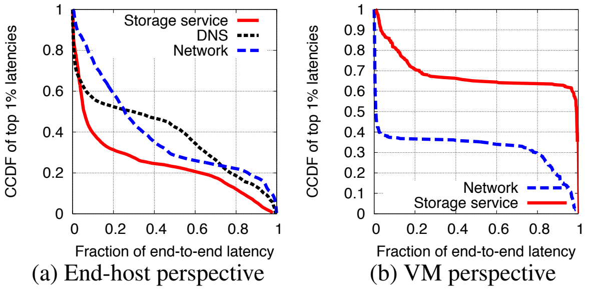


Figure 2: *(a) Absolute and (b) relative inflation in median user-level request latency with respect to ideal latency. Note logscale on x-axis in both graphs.*



(a) End-host perspective (b) VM perspective

Figure 3: *Breakdown of components of tail latencies.*

We setup clients on PlanetLab nodes and applications on S3, emulate interactions between users and applications using real world traces [6, 30], and measure the page load time/sync completion time.

Ideally, with no latency variance, in the webpage application, page load time should be the same as the latency of fetching a single object if clients fetch all objects on the page in parallel, and in the social network application, the sync completion time should be the same as the latency incurred by the farthest follower to fetch a single object. However, Figure 2 shows that, for over 80% of PlanetLab nodes, latency variance causes at least 50ms latency inflation in the *median* page load time and at least 100ms latency inflation in the *median* sync completion time. This corresponds to a 2x relative inflation for more than 50% of users.

Causes for tail latencies. We observe two characteristics that dictate which solutions can potentially reduce the tail of these latency distributions.

First, we find that neither are the top 1% of latency samples clustered together in time nor are they correlated with time of day. Thus, the tail of the latency distribution is dominated by isolated spikes, rather than sustained periods of high latencies. *Therefore, a solution that monitors load and reacts to latency spikes will be ineffective.*

Second, Figure 3(a) shows that all three components of end-to-end latency significantly influence tail latency values. DNS latency, network latency, and latency within the storage service account for over half the end-to-end latency on more than 40%, 25%, and 20% of tail latency samples. Since network latencies as measured from PlanetLab nodes conflate latencies over the Internet and within the cloud service's data center network, we also study the composition of tail latencies as seen in our measurements from VMs to the local storage service. In this case too, Figure 3(b) shows that both components of end-to-end latency—latency within the storage service, and latency over the data center network—contribute significantly to a large fraction of tail latency samples. *Thus, any solution that reduces latency variance will have to address all of these sources of latency spikes.*

3 Overview of CosTLO

Goal. We design CosTLO to meet any application's service-level objectives (SLOs) for the extent to which it seeks to reduce latency variance for its users. To ensure that CosTLO is broadly applicable across several classes of applications, we consider the most fundamental SLO that applications can build upon—SLOs that bound the variance of the latencies of individual PUT/GET operations; we discuss CosTLO's ability to handle more complex application-specific SLOs in Section 6.

Though there are several ways in which such SLOs can be specified, we do not consider SLOs that bound the absolute value of, say, 99^{th} percentile GET/PUT latency; due to the non-uniform geographic distribution of data centers, a single bound on tail latencies for all end-hosts will not help reduce latency variance for end-hosts with proximate data centers. Instead, we focus on SLOs that limit the tail latencies for any end-host *relative* to the latency distribution experienced by *that* end-host. Specifically, we consider SLOs which bound the difference, for any end-host, between 99^{th} percentile latency and its baseline median latency (i.e., the median latency that it experiences without CosTLO). Every application specifies such a bound separately for GETs and PUTs.

Approach. Since tail latency samples are dominated by isolated spikes, our high-level approach is to augment any GET/PUT request with a set of redundant requests, so that the first response can be considered. Though this is a well-known approach for reducing tail latencies [38, 22, 13], CosTLO is unique in exploiting several ways of issuing redundant requests in combination.

For example, consider downloads from the closest S3 data center at the PlanetLab node in University of Kansas. When this client fetches objects by issuing single GET requests, the difference between the 99^{th} percentile and median latencies is 214ms. The simplest way to reduce variance is to have the client issue two concurrent GET requests to download an object (Figure 4(a)). This decreases the gap between 99^{th} percentile and baseline median latency to 110ms, but doubles the cost for GET operations and network bandwidth. Alternatively, the client can issue a single GET request to a VM in the cloud, which can in turn issue two concurrent requests

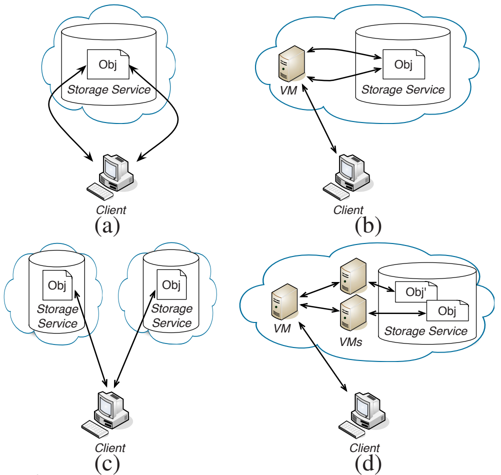


Figure 4: *Illustration of various ways in which CosTLO can concurrently issue requests: (a) to a single object in a storage service, (b) to a single object via a relay VM, (c) to storage services in multiple data centers, or (d) via multiple relay VMs.*

for the requested object to the local storage service (Figure 4(b)). While this adds VM costs and the 99^{th} percentile latency is now 135ms higher than the baseline median latency, relaying redundant requests via VMs reduces bandwidth costs (since a single copy of the object leaves the data center). A third option is to have the client concurrently fetch copies of the object from multiple data centers (Figure 4(c)), e.g., the two closest S3 data centers. This strategy—the best of the three options in terms of reducing variance (inflation in 99^{th} percentile compared to baseline median drops to 34ms)—eliminates the overhead of VM costs but increases storage costs.

Challenges. This example illustrates how various forms of redundancy differ in the tradeoff between reducing variance and increasing cost. Choosing from these various options, so as to satisfy an application's SLO cost-effectively, is challenging for several reasons.

- ***Large configuration space.*** There exist an unbounded number of configurations in which CosTLO can issue redundant requests. This is not only because the degree of parallelism is unbounded, but also because different types of redundancy can be combined with each other. For example, Figure 4(d) shows a configuration that both 1) uses multiple relay VMs to route around latency spikes in the data center network, and 2) issues requests to different objects that are copies of each other. This unbounded configuration space makes it impossible to simply *measure* the latency distribution offered by every candidate configuration of CosTLO.
- ***Complex service architectures.*** However, *predicting* the impact on latencies of any particular approach for issuing redundant requests is complicated by the fact that we have little visibility into the architecture of any cloud storage service. As we describe later, due to correlations between concurrent requests, we cannot estimate the latencies obtained with k concurrent requests simply by considering the minimum of k independent samples of a single request's latency distribution.
- ***Multi-dimensional pricing policies.*** Finally, minimizing CosTLO's cost overhead is made complex by the fact that cloud services charge customers based on a combination of storage, request, VM, and bandwidth costs. Each of the potential ways in which redundant requests can be issued impacts a subset of these pricing dimensions, and the extent to which it does so depends on the application's workload.

4 Characterizing Configuration Space

CosTLO's approach of issuing redundant requests to reduce tail latencies can broadly be applied in two ways. One way is to concurrently issue the same request multiple times in order to *implicitly* exploit load balancing in the Internet or inside cloud services. For example, issuing multiple GET requests concurrently to the same object may lower latencies either because different requests take different paths through the Internet to the same data center, or because different requests may be served by different storage servers that host replicas of the same object. An alternate way is to *explicitly* enforce diversity by concurrently issuing a set of requests that differ from each other, yet have the same effect, e.g., by storing multiple copies of an object and issuing concurrent requests to different copies, or by issuing concurrent requests to different front-ends of a storage service.

Here, we empirically evaluate on both S3 and Azure the efficacy of several approaches for reducing tail latencies in three components of end-to-end latency: Internet latency, data center network latency, and latency in the storage service. We ignore DNS latency since applications often do not control how clients perform DNS lookups and concurrently querying multiple nameservers to reduce DNS latencies has no impact on cost.

4.1 Internet latencies

To examine the utility of different approaches on reducing Internet tail latencies, we issue pairs of concurrent GET requests from each PlanetLab node in three different ways and then compare the measured tail latencies with those seen with single requests. We use the notation "nx $C[m]$" to denote a setting in which every PlanetLab node issues n concurrent requests to its m^{th} closest data center in cloud C, where C is either S3, Azure, or the union of data centers in the two ("S3/Azure").

Multiple requests to same data center. To account for spikes in Internet latency, we first consider every end-host concurrently issuing multiple requests to the storage service in the data center closest to it. Load balancing in the Internet [14] may result in concurrent requests taking

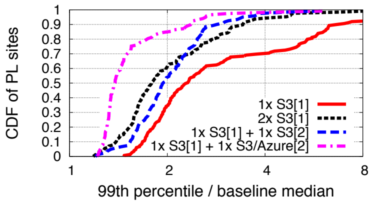


Figure 5: *Impact on Internet tail latencies of different ways to send two concurrent requests. Logscale on x-axis.*

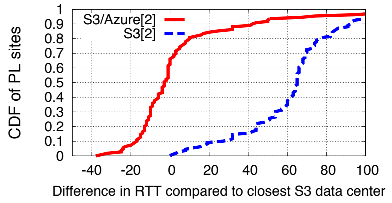


Figure 6: *Comparison of second closest data center within a cloud service and across cloud services.*

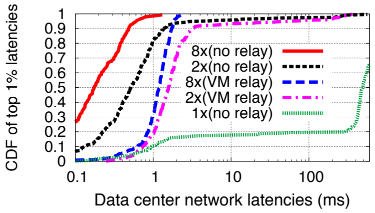


Figure 7: *Different ways of exploiting path diversity in the data center network. Note logscale on x-axis.*

ing different paths to the same data center.[1] However, as shown by the *"2x S3[1]"* line in Figure 5, though issuing two concurrent requests to the same data center does reduce the inflation in tail latencies, relative inflation seen at the median PlanetLab node remains close to 2x; the *"1x S3[1]"* line represents the baseline where end-hosts issue single GET requests to their closest data center.

Requests to multiple data centers. Since path diversity to the same data center is insufficient to tame Internet latency spikes, we next consider issuing concurrent requests to multiple data centers; in addition to a GET request to its closest S3 data center, we have every node issue a GET request in parallel to its second closest S3 data center. The *"1x S3[1] + 1x S3[2]"* line in Figure 5 shows that this strategy offers little benefit in reducing latency variance. This is because, for most PlanetLab nodes, the second closest data center is too far to help tame latency spikes to the node's closest data center.

The root cause for this is that any particular cloud service provider provisions its data centers in a manner that maximizes geographical coverage. Hence, any pair of data centers in the same cloud service are distant from each other. For example, the *"S3[2]"* line in Figure 6 shows that RTT to the second closest data center in S3 is 40ms greater than the RTT to the closest S3 data center for over 80% of PlanetLab nodes.

Leveraging multiple cloud providers. Though a single cloud provider's data centers are distant from each other, we observe that different cloud providers often have nearby data centers. For example, Figure 6 shows that, for over 80% of PlanetLab nodes, RTT to the second closest data center across S3 and Azure is within 25ms of the RTT to the closest S3 data center.

Therefore, leveraging the fact that storage services offered by all cloud providers largely offer the same PUT/GET interface, every client of an application can download copies of an object in parallel from 1) the closest data center among the ones on which the application is deployed, and 2) the second closest data center across all storage services that offer a PUT/GET interface. Figure 5 shows that doing so reduces the inflation in 99^{th} percentile GET latency to be less than 1.5x the baseline median at 70% of PlanetLab nodes. Note that the application itself can be deployed across a single cloud provider's data centers. As we describe later (Section 5.1), CosTLO can maintain copies of objects without the application's knowledge.

4.2 Data center network latencies

Next, we consider strategies for tackling latency spikes within a cloud service's data center network.

In this case, we first attempt to implicitly exploit path diversity by issuing the same PUT/GET request multiple times in parallel from a VM to the local storage service. Load balancing within the data center network [24] may cause concurrent requests to take different routes to the same front-end of the storage service, thus enabling us to avoid latency spikes that occur on any one path.

Alternatively, we can explicitly exploit path diversity in two ways. When a VM issues a GET/PUT to the local storage service, we can either relay each request through a different VM (Figure 4(d)), or issue each request to a different front-end of the storage service. While the latter approach is applicable in S3, all requests issued by the same tenant are submitted to the same front-end [20] in Azure. Therefore, we only consider here the former way of explicitly exploiting path diversity.

In one of Azure's data centers, Figure 7 compares the distribution of tail latencies over the network in three scenarios for how a VM downloads objects from the local storage service: 1) a single request is issued, 2) concurrent requests are issued directly to the same front-end, and 3) concurrent requests are relayed via different VMs. In the latter two cases, we experiment with different levels of parallelism. We see that both implicit and explicit exploitation of path diversity significantly reduce tail latencies, with higher levels of parallelism offering greater reduction. However, using VMs as relays adds some overhead, likely due to requests traversing longer routes.

4.3 Storage service latencies

Finally, we evaluate two approaches for reducing latency spikes within the storage service, i.e., latency between when a request is received at a front-end and when it sends back the response. When issuing n concurrent requests to a storage service, we either issue all n requests for the same object or to n different objects. The former attempts to implicitly leverage the replication of

[1] Multiple requests may also help in surviving packet losses. However, loss rates in our measurements are below 0.1%, thus making them an insignificant factor in causing latency spikes.

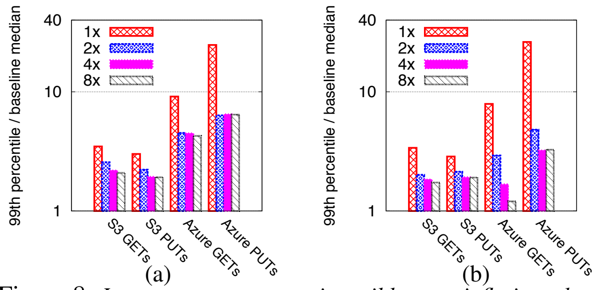


Figure 8: *Impact on storage service tail latency inflation when issuing concurrent requests to (a) the same object, and (b) to different objects. Note logscale on y-axis.*

objects within the storage service, whereas the latter explicitly creates and utilizes copies of objects. In either case, if concurrent requests are served by different storage servers, latency spikes at any one server can be overridden by other servers that are lightly loaded.

At one data center each in Azure and S3, Figure 8 shows that both approaches for issuing concurrent requests significantly reduce tail GET and PUT latencies. However, the takeaways differ between Azure and S3. On S3, irrespective of whether we issue multiple requests to the same object or to different objects, the reduction in 99^{th} percentile latency tails off with increasing parallelism. As seen later in Section 5, this is because, in S3, concurrent requests from a VM incur the same latency over the network, which becomes the bottleneck in the tail. In contrast, on Azure, 99^{th} percentile GET latencies do not reduce further when more than 2 concurrent requests are issued to the same object, but tail GET latencies continue to drop significantly with increasing parallelism when concurrent requests are issued to different objects. In the case of PUTs, the benefits of redundancy tail off at parallelism levels greater than 2 due to Azure's serialization of PUTs issued by the same tenant [20].

4.4 Takeaways

In summary, our measurement study highlights the following viable options for CosTLO to reduce latency variance via redundancy. First, CosTLO can tackle spikes in Internet latencies by issuing multiple requests to a client's closest data center. If greater reduction in Internet tail latencies is desired, CosTLO must concurrently issue requests to the two closest data centers to the client from the union of data centers in multiple cloud services. Second, for latency spikes in a data center's network, it suffices to issue multiple requests to the storage service in that data center. While explicitly relaying requests via VMs may help reduce bandwidth costs (as seen in our example earlier in Section 3), they do not offer additional benefits in reducing latencies. Finally, for latency spikes within the storage service, CosTLO can issue multiple requests either to the same object or to different objects that are all copies of the object being accessed.

5 Cost-effective Support for SLOs

Next, we describe how CosTLO combines the use of the above-mentioned viable redundancy options in order to satisfy an application's SLO cost-effectively.

5.1 System architecture

Application interface. As shown in Figure 9(a), application code on end-hosts links to CosTLO's client library and uses the GET operation[2] in this library to fetch data from cloud storage. The client library issues a set of GET requests to download an object and returns the object's data to the application as soon as any one GET completes. Unlike downloads, we let client-side application code upload data to its own VMs, because the application may need to update application-specific metadata before writing user-uploaded data to cloud storage. The application code in these VMs links to CosTLO's VM library and invokes the PUT operation in this library to write data to the local storage service. The VM library in turn issues a set of PUT requests to the local storage service, and informs the application that the PUT operation is complete once any one of the PUT requests finish. CosTLO offers the same consistency semantics as S3 [3]: read-after-write consistency for PUTs of new objects and eventual consistency for overwrite PUTs; we discuss how CosTLO can support strong consistency later in Section 7.

Configuration selection. CosTLO's central ConfSelector selects the configuration in which its client library and VM library should serve PUTs and GETs. ConfSelector divides time into epochs, and at the start of every epoch, it selects a new configuration separately for every IP prefix, since Internet latencies to any particular data center are similar from all end-hosts in a prefix [31]. To exploit weekly stability in workloads [12], we set epoch durations to one week; we do not consider exploiting diurnal workload patterns because we observe good cost-efficiency even when only leveraging weekly workloads stability. At the start of every epoch, the CosTLO library on every end-host and instances of CosTLO's VM library in every data center fetch the configurations that are relevant to them. Since all objects accessed by a client are replicated as per the configuration associated with the client's prefix, no per-object metadata is necessary. If a client loses its state, it simply re-fetches the configuration in the current epoch for its prefix from ConfSelector.

In the rest of this section, we address three questions: 1) how does ConfSelector identify a cost-effective configuration of CosTLO that can satisfy the application's SLO?, 2) while searching for this cost-effective configuration, how does ConfSelector *estimate* the tail latencies

[2] When ambiguous, we refer to applications invoking CosTLO's GET/PUT *operations*, and CosTLO issuing GET/PUT *requests* to storage services.

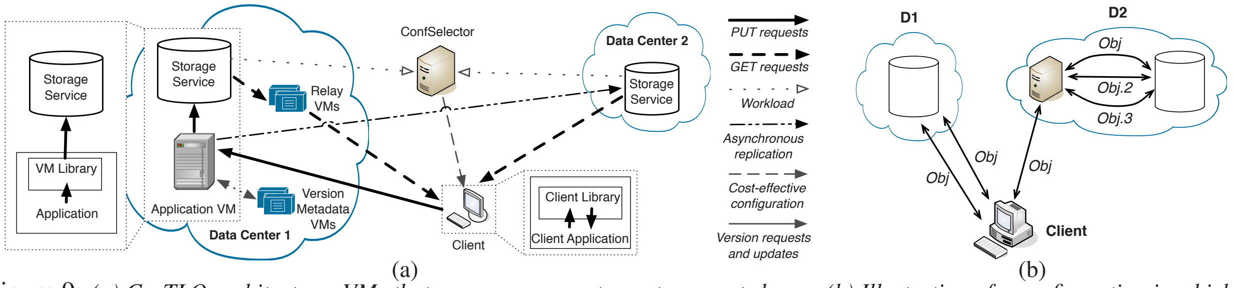


Figure 9: *(a) CosTLO architecture; VMs that run measurement agents are not shown. (b) Illustration of a configuration in which the tuples for data centers D1 and D2 are* (Copies=1, ReqPerCopy=2, VM=False) *and* (Copies=3, ReqPerCopy=1, VM=True). *All edges are annotated with the name of the object for which GET requests are issued when the client requests object Obj.*

for any configuration, given that is impractical to *measure* the latencies offered by every configuration?, and 3) how does CosTLO preserve data consistency?

5.2 Selecting cost-effective configuration

Characterization of workload and cloud services. To estimate the cost overhead and latency variance associated with any CosTLO configuration, ConfSelector 1) takes as input the pricing policies at every data center, 2) uses logs exported by cloud providers to characterize the workload imposed by clients in every prefix, and 3) employs a measurement agent at every data center. Every agent gathers three types of measurements: 1) pings to a representative end-host in every prefix, 2) pairs of concurrent GETs and pairs of concurrent PUTs to the local storage service, and 3) the rates at which VMs can relay PUTs and GETs between end-hosts and the local storage service without any queueing. We ignore the impact of VM failures on tail latency since cloud providers guarantee over 99.95% of uptime for VMs [2, 8].

Representation of configurations. To search through the configuration space, ConfSelector represents every candidate configuration for a prefix as follows. First, a configuration's representation includes two three-tuples, which specify the manner in which end-hosts in the prefix should execute GETs. One three-tuple is for the data center from which the application serves the prefix and another for the data center closest to the prefix among all other data centers on which CosTLO is deployed. Either tuple specifies 1) number of copies of the object stored in that data center, 2) number of requests issued to each copy, and 3) whether all of these requests are relayed via a VM.[3] Figure 9(b) depicts an example.

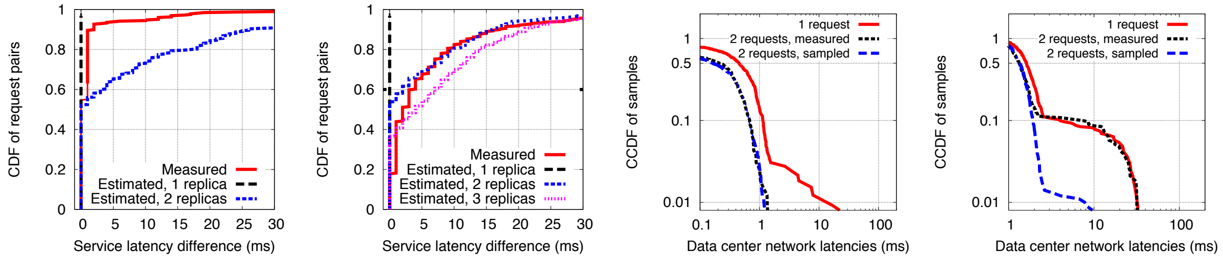
Second, the configuration includes one two-tuple for the manner in which CosTLO's VM library should serve PUTs from the prefix. We use only one tuple in this case, since PUTs from an end-host are served solely at the data center closest to it, and we use a two-tuple, since the VM library does not relay PUTs through other VMs.

Third, to reduce the cost overhead associated with redundant requests, the client/VM library initially issues a single request when serving a GET/PUT. If no response is received for a period, the client/VM library times out and probabilistically issues redundant requests concurrently as specified by the tuples described above. The timeout period ensures that CosTLO's redundancy is focused on requests that incur a high latency, whereas probabilistically issuing redundant requests offers finer-grained control over latency variance. For both PUTs and GETs, the configuration representation specifies the values of the timeout period and probability parameters. Considering the same example from Figure 9(b) but with 70% probability and 50ms timeout period to issue redundant requests, the configuration would be [(1, 2, False), (3, 1, True), 50ms, 70%] (the PUT tuple is ignored here).

Configuration search. Given this representation of the configuration space, ConfSelector identifies a cost-effective configuration of CosTLO for any particular prefix as follows. It initializes the configuration for a prefix to reflect the manner in which an application serves its clients when not using CosTLO—by always issuing only a single request to the data center closest to a client. CosTLO imposes no cost overhead in this configuration.

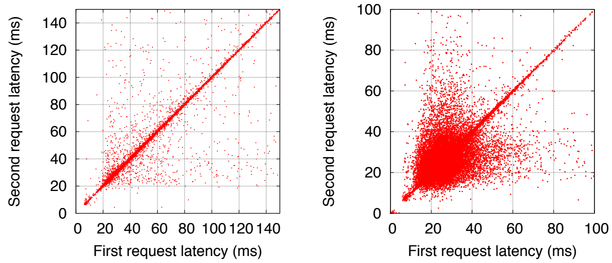
Thereafter, our structured representation of the configuration space enables ConfSelector to step through configurations in the increasing order of cost. For this, ConfSelector maintains a pool of candidate configurations, from which it considers the minimum cost configuration in every step. ConfSelector computes the cost associated with a configuration as the sum of expected costs for storage, VMs, requests, and bandwidth based on the workload for the prefix and the manner in which the configuration mandates that GET/PUT operations be served. If the lowest cost configuration in the current pool does not satisfy the SLO, ConfSelector discards this configuration and inserts all neighbors of this configuration into the pool of candidates. Two configurations are neighbors if they differ in the value of exactly one parameter in the configuration representation. For example, configurations [(1, 2, False), 50ms, 70%] and [(2, 2, False), 50ms, 70%] are neighbors (we only show one GET tuple

[3] If necessary, these three-tuples can be extended to include other dimensions, e.g., whether each request is issued to a different front-end. The dimensions we use here are based on the techniques that we found to be viable in reducing tail latencies on Azure and S3 (Section 4).



(a) (b) (c) (d)

Figure 11: *Distribution of service latency difference between concurrent GET requests offers evidence for GETs to an object (a) being served by one replica in Azure, and (b) being spread across two replicas in S3. Data center network latencies for concurrent requests are (c) uncorrelated on Azure, and (d) correlated on S3. Note logscale on both axes of (c) and (d).*



(a) Azure GETs (b) S3 GETs

Figure 10: *Scatter plot of first vs. second request GET latency when issuing two concurrent requests to a storage service.*

here for simplicity). This process terminates once ConfSelector finds a configuration that satisfies the SLO.

5.3 Estimating latency distribution

To identify when it has found a configuration that will satisfy the application's SLO, for any particular configuration for a prefix, ConfSelector must be able to estimate the latency distribution that clients in that prefix will experience when served in that configuration. For brevity, we present here ConfSelector's estimation of latencies only for GETs, which it computes in four steps. First, for either data center used in the configuration, we estimate the latency distribution when a VM in that data center concurrently issues requests to the local storage service, where the number of requests is specified by the data center's tuple in the configuration representation. Second, we estimate the latency distribution for either data center's tuple by adding the distribution computed above with the latency distribution measured to the prefix from a VM in that data center. Simply adding these distributions works when objects are smaller than 1 KB, and in Section 7, we discuss how to extrapolate this distribution for larger objects. Third, we estimate the client-perceived latency distribution by independently sampling the latency distributions associated with either tuple in the configuration and considering the minimum. Finally, we adjust this distribution to account for the timeout and probability parameters.

The primary challenge here is the first step: estimating the latency distribution when a VM issues concurrent requests to the local storage service. This turns out to be hard due to the dependencies between concurrent requests. While Figure 10 shows the correlation in latencies between two concurrent GET requests to an object at one of Azure's and one of S3's data centers, we also see similar correlations for PUTs and even when the concurrent requests are for different objects. Attempting to model these correlations between concurrent requests by treating the cloud service as a black box did not work well. Therefore, we explicitly model the sources of correlations: concurrent requests may incur the same latency within the storage service if they are served by the same storage server, or incur the same data center network latency if they traverse the same network path.

Modeling replication in storage service. First, at every data center, we use CosTLO's measurements to infer the number of replicas across which the storage service spreads requests to an object. For every pair of concurrent requests issued during CosTLO's measurements, we compute the difference in service latency (i.e., latency within the storage service) between the two requests. We then consider the distribution of this difference across all pairs of concurrent requests to infer the number of replicas in use per object. For example, if the storage service load balances GET requests to an object across 2 replicas, there should be a 50% chance that two concurrent GETs fetch from the same replica, therefore the service latency difference is expected to be 0 half the time. We compare this measured distribution with the expected distribution when the storage service spreads requests across n replicas, where we vary the value of n. We infer the number of replicas used by the service as the value of n for which the estimated and measured distributions most closely match. For example, though both Azure [5] and S3 [4] are known to store 3 replicas of every object, Figures 11(a) and 11(b) show that the measured service latency difference distributions closely match GETs being served from 1 replica on Azure and from 2 replicas on S3.

On the other hand, for concurrent GETs or PUTs issued to different objects, on both Azure and S3, we see that the latency within the storage service is uncorrelated across requests. This is likely because cloud storage services store every object on a randomly chosen server (e.g., by hashing the object's name for load balancing [23]), and hence, requests to different objects are likely to be served by different storage servers.

Modeling load balancing in network. Next, we identify whether concurrent requests issued to the storage service incur the same latency over the data center network, or are their network latencies independent of each other. At any data center, we compute the distribution obtained from the minimum of two independent samples of the measured data center network latency distribution for a single request. We then compare this distribution to the measured value of the minimum data center network latency seen across two concurrent requests.

Figure 11(c) shows that, on Azure, the distribution obtained by independent sampling closely matches the measured distribution, thus showing that network latencies for concurrent requests are uncorrelated. Whereas, on S3, Figure 11(d) shows that the measured distribution for the minimum across two requests is almost identical to the data center network latency component of any single request; this shows that concurrent requests on S3 incur the same network latency.

Estimating VM-to-service latency. Given these models for replication and load balancing, we estimate the end-to-end latency distribution as follows when a VM issues k concurrent requests to the local storage service. If concurrent requests are known to have the same latency over the service's data center network, we sample the measured data center network latency distribution once and use this value for all requests; if not, we independently sample once for each request. If all k requests are to the same object, then we randomly assign every request to one of the replicas of the object, where the number of replicas is identified as described above. If the k requests are for k different objects, then we assume that no two requests are served from the same storage server. In either case, for each storage server, we independently choose a sample from the service latency distribution for a single request and assign that to be the service latency for all requests assigned to that server. Finally, for each of the k requests, we sum up their assigned data center network latency and service latency values, and estimate the end-to-end latency at the VM as the minimum of this sum across the k requests.

Note that our latency estimation models may potentially break down at high storage service load. But, we have not seen any evidence of this so far, since we see the same latency distribution irrespective of whether we issue requests once every 3 seconds or once every 200ms.

5.4 Ensuring data consistency

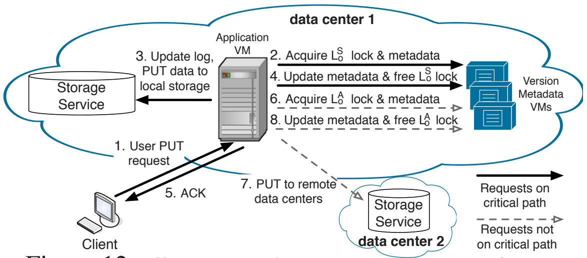

data center 1
Application VM
3. Update log, PUT data to local storage
Storage Service
2. Acquire L_o^S lock & metadata
4. Update metadata & free L_o^S lock
6. Acquire L_o^A lock & metadata
8. Update metadata & free L_o^A lock
Version Metadata VMs
1. User PUT request
5. ACK
Client
7. PUT to remote data centers
Storage Service
data center 2
Requests on critical path
Requests not on critical path

Figure 12: *Illustration of CosTLO's execution of PUTs.*

CosTLO can afford to inform the application that a PUT operation is complete as soon as any of the PUT requests that it issues to serve the operation finish, because the underlying cloud services guarantee that the data written by a completed PUT request will be durable [5, 4]. However, this design decision makes it challenging for CosTLO to ensure that, eventually, all GETs for an object will consistently return the same data. First, if the application issues back-to-back or concurrent PUT operations on the same object, redundant PUT requests that are still pending from a completed PUT operation may potentially overwrite updates written by subsequent PUT operations. Second, if an application VM restarts after only a subset of the PUT requests issued to serve a PUT operation complete, the VM library will not realize if some of the remaining PUT requests fail, thus causing some of the copies of the object to potentially not reflect the latest update to the object.

Figure 12 illustrates the execution of PUTs in CosTLO accounting for these concerns. In every data center, CosTLO maintains a set of VMs that store in memory (with a persistent backup) the latest version number and the status of two locks L_o^S and L_o^A for every object o stored in that data center. We use L_o^S for synchronous PUTs to local storage service and L_o^A for asynchronous PUTs to remote storage services. When serving a PUT operation on object o, the VM library first queries the local cluster of CosTLO's VMs to obtain lock L_o^S and learn o's current version. Once it acquires the lock, the library appends to a persistent log (maintained locally on the VM) the update that needs to be written to o and all the PUT requests that the library needs to issue as per the configuration for the client issuing this PUT operation. By appending the status of every response to the log, the library ensures that it knows which PUTs to re-issue, even across VM restarts. Once all PUT requests complete, the library releases lock L_o^S, updating o's version in the process. At some point later, the library attempts to acquire lock L_o^A, and if o's version has not changed by then, it updates the remaining copies of o and subsequently releases the lock. If o's version has changed, the library just needs to release the lock, since there exists a newer PUT operation on this key and that PUT's asynchronous propagation will suffice to update the remaining copies of o.

Note that, since the application is unaware of the replication of objects across data centers, all PUT operations on an object will be issued by the application's VMs in

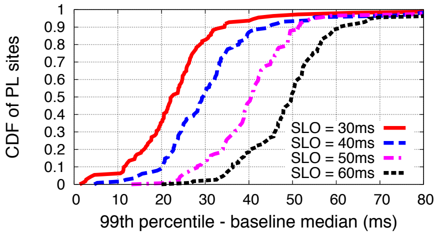


Figure 13: *Verification of CosTLO's ability to satisfy SLOs.*

the same data center. Hence, the VM library needs to acquire locks only from CosTLO's VMs within the local data center, thus ensuring that locking operations add negligible latency. Also note that, when an application issues back-to-back PUT operations, execution of the latter PUT has to wait for the lock L_o^S (for the object o being updated) to be released. This can potentially increase[4] tail latencies if multiple PUT requests need to complete before L_o^S is released. Therefore, in the rare case when an application often issues back-to-back or concurrent PUTs for the same object, the application should choose an SLO that offers no improvement in PUT latency variance; this will ensure that CosTLO executes any PUT operation by issuing a single PUT request.

6 Evaluation

We evaluate CosTLO from three perspectives: 1) its ability to satisfy latency SLOs, 2) its cost-effectiveness in doing so, and 3) its efficiency in various respects. We perform our evaluation from the perspective of an application deployed across all of Amazon's data centers. We deploy CosTLO across Azure's and S3's data centers, and use PlanetLab nodes at 120 sites as clients.

6.1 Ability to satisfy SLOs

SLOs on individual operations. To verify CosTLO's ability to satisfy latency SLOs, we mimic a deployment of Wikipedia using server-side logs of objects requested from the English version of Wikipedia [6]. We randomly select a 1% sample from the datasets for two consecutive weeks. We provide the workload from the first week to ConfSelector as input, and have it select cost-effective configurations for 120 PlanetLab nodes. We then run CosTLO with every node configured in the manner selected by ConfSelector. We replay the workload from the second week, with every GET request assigned to a random PlanetLab node. We repeat this experiment for four SLO values—30ms, 40ms, 50ms, and 60ms. In all cases, since we issue GETs/PUTs to S3 and Azure, our measurements are affected by Internet congestion and by contention with S3's and Azure's customers.

Figure 13 shows the distribution of the measured difference between the 99^{th} percentile and baseline median latencies at every PlanetLab node. For all SLOs, the latency variance delivered by CosTLO is within the input SLO on most nodes; without CosTLO, the difference between 99^{th} percentile and baseline median GET latencies is greater than 60ms for 75% of PlanetLab nodes (Figure 1(a)). Latency variance with CosTLO is, in fact, well below the SLO in many cases; due to discontinuous drops in the latency distribution across neighboring configurations, as ConfSelector steps through the configuration space, it often directly transitions from a configuration that violates the SLO to one that exceeds it.

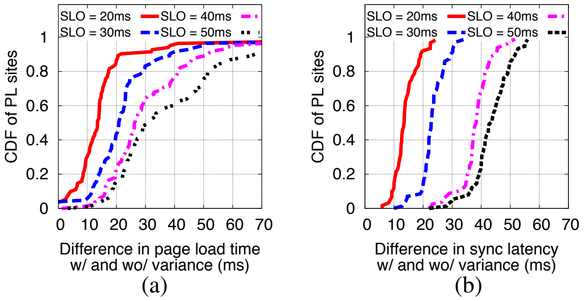


Figure 14: *CosTLO's ability to satisfy application-specific SLOs for (a) webpage and (b) social network applications.*

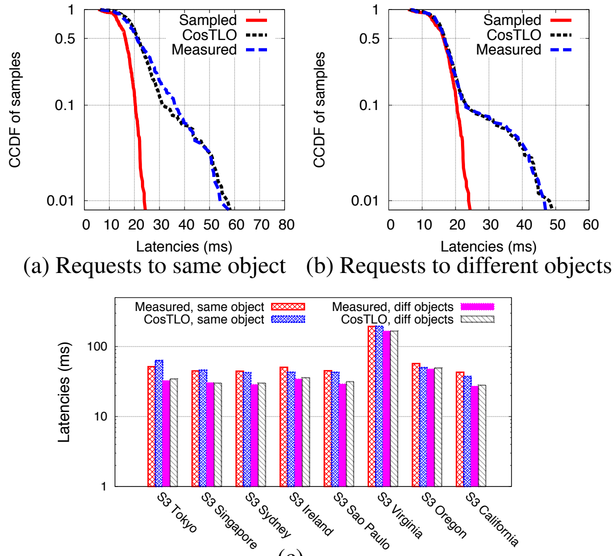
Note that, though we only demonstrate CosTLO's ability to satisfy GET latency SLOs here (because the trace from Wikipedia only contains GETs), CosTLO can also reduce the latency variance for PUTs as described earlier. In contrast, in-memory caching of data can only reduce tail latencies for GETs, but not for PUTs.

Application-specific SLOs. CosTLO's design is easily extensible to handle application-specific SLOs, rather than the SLOs for the latencies of individual PUT/GET operations. Here, we show the results of using CosTLO to reduce user-perceived latencies in the two applications from Section 2. In the webpage application, we modify ConfSelector so that it uses the models in Section 5.3 to estimate the distribution for the latency incurred when the client library fetches 50 objects in parallel and waits for at least one GET to each of these objects to complete. In the social network application, since we need to estimate latencies from multiple users, we extend the configuration representation in ConfSelector such that it contains the configuration tuples of all of a user's followers. The sync completion time is determined when all followers have at least one GET completed. We use this modified version of ConfSelector to select configurations for all PlanetLab nodes and run CosTLO's client library on every node as per these configurations. Figure 14 shows that CosTLO is able to satisfy application-specific SLOs in both applications.

6.2 Accuracy of estimating latency distributions

CosTLO is able to meet latency SLOs due to its accurate estimation of the end-to-end latency distributions in any configuration. Our simple approaches of considering

[4] Note that we can reduce the extent of this increase in inflation by having CosTLO maintain a lock $L_{o,c}^S$ for every copy c of object o, but we do not present such a design here to keep the discussion simple.



(a) Requests to same object (b) Requests to different objects

(c)

Figure 15: *Accuracy of estimating GET latency distribution for 8 concurrent GET requests from VM to local storage service. (a and b) Comparison of latency distributions in one S3 region. (c) Comparison across all S3 regions of 99^{th} percentile latencies. Note logscale on y-axis of all three graphs.*

the minimum of the latency distributions across data centers and of adding VM-to-prefix and VM-to-service latency distributions work reasonably well; in either case, CosTLO's estimates show less than 15% error for 90% of PlanetLab nodes. Therefore, here we focus on demonstrating the accuracy of our estimation of the latency distribution when a VM concurrently issues a set of requests to the local storage service. Recall that CosTLO only gathers measurements when issuing pairs of concurrent requests. We evaluate its ability to estimate the latency distribution for higher levels of parallelism.

Figures 15(a) and 15(b) compare the measured and estimated latency distributions when issuing 8 concurrent GETs from a VM to the local storage service; all concurrent requests are for the same object in the former and to different objects in the latter. In both cases, our estimated latency distribution closely matches the measured distribution, even in the tail. In contrast, if we estimate the latency distribution for 8 concurrent GETs by independently sampling the latency distribution for a single request 8 times and considering the minimum, we significantly under-estimate the tail of the distribution. Additionally, Figure 15(c) shows that the relative error between the measured and estimated values of the 99^{th} percentile GET latency is less than 5% in the median S3 region; latencies are higher for S3's Virginia data center because it is the most widely used data center in S3.

6.3 Cost-effectiveness

An application that uses CosTLO incurs additional costs for storing copies of objects, for operations and bandwidth due to redundant requests, and for VMs used either as relays or to manage locks and version numbers. We again use Wikipedia's workload to quantify this overhead on an application provider's costs.

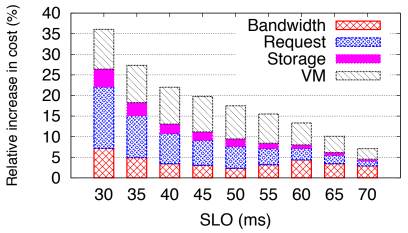
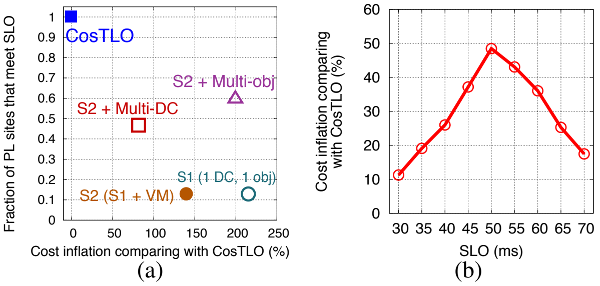


Figure 16: *CosTLO's cost-effectiveness in satisfying SLOs.*



(a) (b)

Figure 17: *(a) Utility of CosTLO's components in reducing cost and meeting SLO = 30ms. (b) Cost inflation when not using timeout and probability parameters.*

Figure 16 shows the relative cost overhead as a function of the latency SLO, with the cost split into its four components. At the higher end of the examined range of SLO values, CosTLO caps tail latency inflation at 70ms—which is less than the inflation observed at the median node when not using CosTLO—with less than 8% increase in cost. As the SLO decreases, i.e., as lower variance is desired, cost increases initially due to an increase in the number of redundant requests. Thereafter, as the SLO further decreases, CosTLO begins to use more relay VMs so that only one copy of any requested object leaves the data center, thus decreasing bandwidth costs at the expense of VM costs. As the SLO decreases further, CosTLO begins concurrently issuing requests to multiple data centers, thus again increasing bandwidth costs. Storage costs and cost for VMs that manage locks and version numbers remain low for all SLO values, because 1) on both Amazon's and Microsoft's cloud services, storage is significantly cheaper than GET/PUT requests, VMs, and network transfers, and 2) lock status and version numbers for all 70M objects in the English version of Wikipedia fit into the memory of a small instance VM, which costs less than $20 per month on EC2.

6.4 Utility of CosTLO's components

CosTLO's ability to satisfy SLOs cost-effectively crucially depends on its *combined use* of various forms of issuing redundancy. We illustrate this in Figure 17(a) by comparing CosTLO with several strategies that each use a subset of the dimensions in CosTLO's configuration space. For each strategy, we compute the fraction of PlanetLab nodes for which it is able to satisfy an SLO of 30ms, and across the nodes on which the strategy does meet the SLO, we compare its cost with CosTLO's.

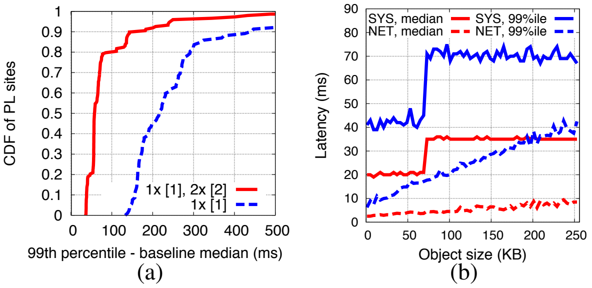
First, the simplest strategy $S1$, which only issues redundant requests to a single copy of any object in the data center closest to any client, can meet the SLO on only a little over 10% of nodes. Adding the use of relay VMs ($S2$) reduces cost inflation compared to CosTLO from over 200% to less than 150%, but the ability to meet the SLO remains unchanged. We can improve the ability to satisfy the SLO by adding the option of issuing redundant requests either to multiple copies of every object in the closest data center or to multiple data centers. However, the fraction of nodes on which the SLO can be met remains below 60% if we use one of these two options. Only by combining the use of relay VMs, multiple copies of objects, and multiple data centers is CosTLO able to meet the SLO at all nodes, at significantly lower cost.

In addition, we illustrate the utility of CosTLO waiting for a timeout period before issuing redundant requests and issuing redundant requests probabilistically. For every SLO in the range 30ms to 70ms, Figure 17(b) compares CosTLO's cost overhead when it uses the timeout and probability parameters versus when it does not. The cost overhead of not using the timeout and probability parameters is low when the SLO is extremely low or extremely high. In the former case, most PlanetLab nodes need to issue redundant requests at all times without any timeout in order to meet the SLO, whereas in the latter case, the SLO is satisfied for most PlanetLab nodes even without redundant requests. However, for many intermediate SLO values—that are neither too loose nor too stringent—not using the timeout and probability parameters increases cost significantly, by as much as 48%.

6.5 Efficiency

Measurement cost. The cost associated with CosTLO's measurements depends on the number of latency samples necessary to accurately sample latency distributions. To quantify the stationarity in latencies, we consider a dataset of 200K latency measurements gathered over a week from VMs in every S3 and Azure data center. We then consider subsets of these datasets, varying the number of samples considered. In all datasets, we find that 10K samples are sufficient to obtain a reasonably accurate value of the 99^{th} percentile latency. In the ping, GET, and PUT latency measurements, the 99^{th} percentile from a subset of 10K samples was off from the 99^{th} percentile in the entire dataset by only 2.9%, 3.8%, and 2.2% on average.

Thus, at every data center, CosTLO's weekly measurement costs include: 1) 20K GETs and PUTs (since CosTLO gathers data with pairs of concurrent requests), 2) 10K pings to every end-host prefix, and 3) one "small instance" VM (which is sufficient to support this scale of measurements). Accounting for the roughly 120K IP prefixes at the Internet's edge [27], 8 S3 data centers, and 13 Azure data centers, these measurements translate to a total cost of $392 per week. These minimal measurement costs are shared across all applications that use CosTLO.



(a) (b)

Figure 18: *(a) CosTLO's utility in reducing latency variance when offering strong consistency; 1 PUT request per copy. (b) latency breakdown for objects of different sizes.*

Configuration selection runtime. We run ConfSelector to select the configurations for 120 PlanetLab nodes, and we compute the average runtime per node. We repeat this for SLO values ranging from 20ms to 100ms. Extrapolating the average runtime per node, we estimate that, for all SLO values, ConfSelector needs less than a day to select the configuration for all 120K edge prefixes on a server with 16 cores. Hence, ConfSelector can identify the configurations for a particular week during the last day of the previous week. Moreover, since ConfSelector independently selects configurations for different prefixes, this runtime is easily reduced by parallelizing ConfSelector's execution across a cluster of servers.

7 Discussion

Strong consistency. Many applications (e.g., Google Docs) require their underlying storage to offer strong consistency. For such applications, CosTLO uses only strongly consistent storage services, e.g., it can use Azure but not S3. In addition, two modifications are necessary in the execution of a PUT operation on any object o. First, to ensure linearizability of PUTs, the VM library synchronously updates all copies of o before releasing lock L_o^S. Second, instead of the library informing the application when any one PUT request completes, the application registers for two callbacks—1) *quorumPUTsDone*, for when at least one PUT request each completes on a quorum of o's copies, and 2) *allPUTsDone*, when all PUTs finish. The *quorumPUTsDone* callback indicates to the application that subsequent GET operations on o will fetch the latest version, if the client library waits for responses from a quorum of copies when serving GETs.

After these changes, Figure 18(a) shows the PUT latency variance offered by CosTLO when every object accessed by a PlanetLab node has one copy and two copies, respectively, in the closest Azure data center and the second closest data center across S3 and Azure; for this analysis, we ignore that S3 does not offer strong consistency. Despite having to wait for PUT requests on a quorum of copies to complete, and though a quorum of every object's copies are stored in a different data center than the application VMs that issue PUT operations on the object, CosTLO more than halves the PUT latency inflation for the median node. This again highlights the utility of redundant copies and requests, and of CosTLO's use of multiple cloud services.

Latency estimation for larger objects. One can potentially extend CosTLO's approach for estimating latency variance to larger objects as follows. We conduct measurements on objects from 1 KB to 256 KB when issuing one GET request at a time to each object from a local VM, and Figure 18(b) shows the results from one data center. We see that network latency is proportional to object size, and storage service latency is a step function of object size. Although different data centers may have different step functions (we observe that some data centers have the same storage service latency distribution for all sizes in the 256 KB range), the smallest range that has a fixed storage service latency distribution is until 64 KB, which is a typical block size in distributed storage systems [21]. Therefore, to estimate latencies for objects of different sizes, we can leverage the fact that objects with the same number of blocks have the same storage service latency distribution.

Scale of adoption. CosTLO's approach of issuing redundant requests makes it unviable if all applications adopt it. However, we believe that increasing adoption of CosTLO will emphasize the demand for latency SLOs and spur cloud providers to suitably modify their services. In the interim, CosTLO minimizes the cost overhead incurred by application providers who seek to improve predictability in user-perceived latencies without having to wait for any changes to cloud services. Moreover, cloud service providers have little control over reducing variance in the latency on the Internet path between end-hosts and their data centers.

8 Related Work

Redesigning cloud services. Several recent proposals redesign storage systems and data centers to improve tail latency performance [36], to offer bandwidth guarantees to tenants [15, 33, 16, 37], or to ensure predictable completion times for TCP flows [42, 26, 40]. However, all of these proposals require modifications to a cloud service's infrastructure. It is unclear when, and if, cloud services will revamp their infrastructure to these more complex architectures. CosTLO instead satisfies latency SLOs for applications deployed on the cloud without having to wait for any modifications to cloud services.

Reducing tail latencies. The approach of issuing redundant requests to reduce tail latencies has been considered previously [22, 38], but the focus has primarily been on understanding the implications of redundancy on system load. In contrast, our work demonstrates how the approach of using redundant requests should be applied in the context of cloud storage services, in order to meet latency SLOs while minimizing cost overhead.

Some application providers such as Facebook use in-memory caching of data to reduce tail latencies [32]. However, caching cannot reduce tail latencies associated with PUT requests. Moreover, caching at a single data center cannot tackle latency spikes on Internet paths, and not all application providers will be able to afford caches at multiple data centers that can accommodate enough data to reduce 99^{th} percentile GET latencies.

Cloud measurement studies. Prior studies have compared the performance offered by different cloud providers [29], reverse-engineered cloud service internals [34], and studied application deployments on the cloud [25]. Our measurement study of Azure and S3 is the first to quantify the latency variance on these storage services and to characterize the impact of different forms of redundancy. Moreover, unlike Bodik et al. [18], who focused on characterizing and modeling spikes in application workloads, our measurements show that an application using cloud storage can suffer latency spikes even when there is no spike in that application's workload.

Combining cloud providers. Others have combined the use of multiple cloud providers to improve availability [11, 28], to offer more secure storage [17], and to reduce cost [39, 41]. CosTLO uses cloud storage services offered by multiple providers because 1) the combination offers more data center pairs that are close to each other, and 2) latency spikes on the Internet paths to data centers in different cloud services are uncorrelated.

9 Conclusions

Our measurements of the Azure and S3 storage services highlight the high variance in latencies offered by these services. To enable applications to improve predictability, without having to wait for these services to modify their infrastructure, we have designed and implemented CosTLO, a framework that requires minimal changes to applications. Based on several insights about the causes for latency variance on cloud storage services that we glean from our measurements, our design of CosTLO judiciously combines several instantiations of the approach of issuing redundant requests. Our results show that, despite the unbounded configuration space and opaque cloud service architectures, CosTLO cost-effectively enables applications to meet latency SLOs.

Acknowledgments

This work was supported in part by the National Science Foundation under grants 1463126 and 1150219.

References

[1] 250+ amazing Twitter statistics. http://expandedramblings.com/index.php/march-2013-by-the-numbers-a-few-amazing-twitter-stats/.

[2] Amazon EC2 service level agreement. http://aws.amazon.com/ec2/sla/.

[3] Amazon S3 FAQs. http://aws.amazon.com/s3/faqs/.

[4] Announcing Amazon S3 reduced redundancy storage. http://aws.amazon.com/about-aws/whats-new/2010/05/19/announcing-amazon-s3-reduced-redundancy-storage/.

[5] Azure storage pricing. http://azure.microsoft.com/en-us/pricing/details/storage/.

[6] Page view statistics for Wikimedia projects. http://dumps.wikimedia.org/other/pagecounts-raw/.

[7] Server access log format - Amazon simple storage service. http://docs.aws.amazon.com/AmazonS3/latest/dev/LogFormat.html.

[8] Windows Azure service level agreements. http://azure.microsoft.com/en-us/support/legal/sla/.

[9] Windows Azure storage logging: Using logs to track storage requests. http://blogs.msdn.com/b/windowsazurestorage/archive/2011/08/03/windows-azure-storage-logging-using-logs-to-track-storage-requests.aspx.

[10] Amazon found every 100ms of latency cost them 1% in sales. http://blog.gigaspaces.com/amazon-found-every-100ms-of-latency-cost-them-1-in-sales/, 2008.

[11] H. Abu-Libdeh, L. Princehouse, and H. Weatherspoon. RACS: A case for cloud storage diversity. In *Proceedings of the 1st Annual Symposium on Cloud Computing*, 2010.

[12] S. Agarwal, J. Dunagan, N. Jain, S. Saroiu, and A. Wolman. Volley: Automated data placement for geo-distributed cloud services. In *Proceedings of the 7th USENIX Conference on Networked Systems Design and Implementation*, 2010.

[13] D. G. Andersen, H. Balakrishnan, M. F. Kaashoek, and R. Rao. Improving web availability for clients with MONET. In *Proceedings of the 2nd USENIX Conference on Networked Systems Design and Implementation*, 2005.

[14] B. Augustin, X. Cuvellier, B. Orgogozo, F. Viger, T. Friedman, M. Latapy, C. Magnien, and R. Teixeira. Avoiding traceroute anomalies with Paris traceroute. In *Proceedings of the 6th ACM SIGCOMM Conference on Internet Measurement*, 2006.

[15] H. Ballani, P. Costa, T. Karagiannis, and A. Rowstron. Towards predictable datacenter networks. In *Proceedings of the ACM SIGCOMM Conference*, 2011.

[16] H. Ballani, K. Jang, T. Karagiannis, C. Kim, D. Gunawardena, and G. O'Shea. Chatty tenants and the cloud network sharing problem. In *Proceedings of the 10th USENIX Conference on Networked Systems Design and Implementation*, 2013.

[17] A. Bessani, M. Correia, B. Quaresma, F. Andre, and P. Sousa. DEPSKY: Dependable and secure storage in a cloud-of-clouds. In *Proceedings of the 6th ACM European Conference on Computer Systems*, 2011.

[18] P. Bodik, A. Fox, M. J. Franklin, M. I. Jordan, and D. A. Patterson. Characterizing, modeling, and generating workload spikes for stateful services. In *Proceedings of the 1st Annual Symposium on Cloud Computing*, 2010.

[19] J. Brutlag. Speed matters for Google web search. http://services.google.com/fh/files/blogs/google_delayexp.pdf, 2009.

[20] B. Calder, J. Wang, A. Ogus, N. Nilakantan, A. Skjolsvold, S. McKelvie, Y. Xu, S. Srivastav, J. Wu, H. Simitci, J. Haridas, C. Uddaraju, H. Khatri, A. Edwards, V. Bedekar, S. Mainali, R. Abbasi, A. Agarwal, M. F. u. Haq, M. I. u. Haq, D. Bhardwaj, S. Dayanand, A. Adusumilli, M. McNett, S. Sankaran, K. Manivannan, and L. Rigas. Windows Azure storage: A highly available cloud storage service with strong consistency. In *Proceedings of the 23rd ACM Symposium on Operating Systems Principles*, 2011.

[21] F. Chang, J. Dean, S. Ghemawat, W. C. Hsieh, D. A. Wallach, M. Burrows, T. Chandra, A. Fikes, and R. E. Gruber. Bigtable: A distributed storage system for structured data. In *Proceedings of the 7th Symposium on Operating Systems Design and Implementation*, 2006.

[22] J. Dean and L. A. Barroso. The tail at scale. *Communications of the ACM*, 2013.

[23] G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, and W. Vogels. Dynamo: Amazon's highly available key-value store. In *Proceedings of the 21st ACM Symposium on Operating Systems Principles*, 2007.

[24] A. Greenberg, J. R. Hamilton, 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 *Proceedings of the ACM SIGCOMM Conference*, 2009.

[25] K. He, L. Wang, A. Fisher, A. Gember, A. Akella, and T. Ristenpart. Next stop, the cloud: Understanding modern web service deployment in EC2 and Azure. In *Proceedings of the 13th ACM SIGCOMM Conference on Internet Measurement*, 2013.

[26] C.-Y. Hong, M. Caesar, and P. B. Godfrey. Finishing flows quickly with preemptive scheduling. In *Proceedings of the ACM SIGCOMM Conference*, 2012.

[27] E. Katz-Bassett, H. Madhyastha, J. John, A. Krishnamurthy, D. Wetherall, and T. Anderson. Studying black holes in the Internet with Hubble. In *Proceedings of the 5th USENIX Conference on Networked Systems Design and Implementation*, 2008.

[28] R. Kotla, L. Alvisi, and M. Dahlin. SafeStore: A durable and practical storage system. In *Proceedings of the USENIX Annual Technical Conference*, 2007.

[29] A. Li, X. Yang, S. Kandula, and M. Zhang. CloudCmp: Comparing public cloud providers. In *Proceedings of the 10th ACM SIGCOMM Conference on Internet Measurement*, 2010.

[30] R. Li, S. Wang, H. Deng, R. Wang, and K. C.-C. Chang. Towards social user profiling: Unified and discriminative influence model for inferring home locations. In *Proceedings of the 18th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, 2012.

[31] H. V. Madhyastha, T. Isdal, M. Piatek, C. Dixon, T. Anderson, A. Krishnamurthy, and A. Venkataramani. iPlane: An information plane for distributed services. In *Proceedings of the 7th Symposium on Operating Systems Design and Implementation*, 2006.

[32] R. Nishtala, H. Fugal, S. Grimm, M. Kwiatkowski, H. Lee, H. C. Li, R. McElroy, M. Paleczny, D. Peek, P. Saab, D. Stafford, T. Tung, and V. Venkataramani. Scaling memcache at Facebook. In *Proceedings of the 10th USENIX Conference on Networked Systems Design and Implementation*, 2013.

[33] L. Popa, G. Kumar, M. Chowdhury, A. Krishnamurthy, S. Ratnasamy, and I. Stoica. FairCloud: Sharing the network in cloud computing. In *Proceedings of the ACM SIGCOMM Conference*, 2012.

[34] 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 *Proceedings of the 16th ACM Conference on Computer and Communications Security*, 2009.

[35] S. Souders. Velocity and the bottom line. http://radar.oreilly.com/2009/07/velocity-making-your-site-fast.html, 2009.

[36] L. Suresh, M. Canini, S. Schmid, and A. Feldmann. C3: Cutting tail latency in cloud data stores via adaptive replica selection. In *Proceedings of the 12th USENIX Conference on Networked Systems Design and Implementation*, 2015.

[37] E. Thereska, H. Ballani, G. O'Shea, T. Karagiannis, A. Rowstron, T. Talpey, R. Black, and T. Zhu. IOFlow: A software-defined storage architecture. In *Proceedings of the 24th ACM Symposium on Operating Systems Principles*, 2013.

[38] A. Vulimiri, P. B. Godfrey, R. Mittal, J. Sherry, S. Ratnasamy, and S. Shenker. Low latency via redundancy. In *Proceedings of the 9th ACM Conference on Emerging Networking Experiments and Technologies*, 2013.

[39] A. Wieder, P. Bhatotia, A. Post, and R. Rodrigues. Orchestrating the deployment of computations in the cloud with Conductor. In *Proceedings of the 9th USENIX Conference on Networked Systems Design and Implementation*, 2012.

[40] C. Wilson, H. Ballani, T. Karagiannis, and A. Rowstron. Better never than late: Meeting deadlines in datacenter networks. In *Proceedings of the ACM SIGCOMM Conference*, 2011.

[41] Z. Wu, M. Butkiewicz, D. Perkins, E. Katz-Bassett, and H. V. Madhyastha. SPANStore: Cost-effective geo-replicated storage spanning multiple cloud services. In *Proceedings of the 24th ACM Symposium on Operating Systems Principles*, 2013.

[42] D. Zats, T. Das, P. Mohan, D. Borthakur, and R. Katz. DeTail: Reducing the flow completion time tail in datacenter networks. In *Proceedings of the ACM SIGCOMM Conference*, 2012.

Jitsu: Just-In-Time Summoning of Unikernels

Anil Madhavapeddy, Thomas Leonard, Magnus Skjegstad, Thomas Gazagnaire,
David Sheets, Dave Scott,[1] Richard Mortier, Amir Chaudhry,
Balraj Singh, Jon Ludlam,[1] Jon Crowcroft and Ian Leslie
University of Cambridge, Citrix Systems UK Ltd[1]

Abstract. Network latency is a problem for all cloud services. It can be mitigated by moving computation out of remote datacenters by rapidly instantiating local services near the user. This requires an embedded cloud platform on which to deploy multiple applications securely and quickly. We present *Jitsu*, a new Xen toolstack that satisfies the demands of secure multi-tenant isolation on resource-constrained embedded ARM devices. It does this by using *unikernels*: lightweight, compact, single address space, memory-safe virtual machines (VMs) written in a high-level language. Using fast shared memory channels, Jitsu provides a directory service that launches unikernels in response to network traffic and masks boot latency. Our evaluation shows Jitsu to be a power-efficient and responsive platform for hosting cloud services in the edge network while preserving the strong isolation guarantees of a type-1 hypervisor.

1 Introduction

The benefits of cloud hosting are clear: dynamic resource provisioning, lower capital expenditure, high availability, centralised management. Unfortunately, all services architected and deployed in this way inevitably suffer the same problem: *latency*. Physical separation between remote datacenters where processing occurs and users of these services, imposes unavoidable minimum bounds on network latency. Recent developments in augmented reality (e.g., Google Glass [17]) and voice control (e.g., Apple's Siri) particularly suffer in this regard.

Concurrent with the move of services to the cloud, we are now seeing uptake of the "Internet-of-Things" (IoT), giving rise to our second concern: *integrity*. These devices often rely on the network for their operation but many of the devices we use daily suffer from an unrelenting stream of security exploits, including routers [8], buildings [13] and automobiles [5]. The future success of IoT platforms being deployed in edge networks depends on the convenience of secure multi-tenant isolation that the public cloud utilises.

The widely deployed Xen hypervisor [2] enforces isolation between multiple tenants sharing physical machines. Xen recently added support for hardware virtualized ARM guests, opening up the possibility of building an *embedded cloud*: a system of distributed low-power devices, deployed near users, able to host applications delivering real-time services directly via local networks. There has been a steady increase in ARM boards featuring a favourable energy/price/speed trade-off for constructing embedded systems (e.g., the Cubieboard2 has 1GB RAM, a dual-core A20 ARM CPU and costs £ 39).

We present *Jitsu*, a system for securely managing multi-tenant networked applications on embedded infrastructure. Jitsu re-architects the Xen toolstack to lower the resource overheads of manipulating virtual machines (VMs), overcoming current limitations that prevent Xen from becoming an effective platform for building embedded clouds. Rather than booting conventional VMs, Jitsu services network requests with low latency using unikernels [27] as the unit of deployment. These are small enough to be booted in a few hundred milliseconds, a latency that Jitsu further masks through connection hand-off. The MirageOS unikernels [25] that we use are also secure enough to survive inexpertly managed network-facing deployment.

Jitsu uses the virtual hardware abstraction layer provided by the Xen type-1 hypervisor, adding a new control toolstack that eliminates bottlenecks in VM management (Figure 1). Although developed with unikernels in mind, it preserves sufficient compatibility that many of its benefits apply equally to generic (e.g., Linux or FreeBSD) VMs targeting either ARM or x86.

The specific contributions of this paper are thus: a description of how to build efficient, secure unikernels on the new open-source Xen/ARM (§2); an explanation of the Jitsu Xen toolstack architecture (§3); a comparison of it against other application containment techniques (§4); and finally application deployment scenarios and discussion of the broader lessons learnt (§5).

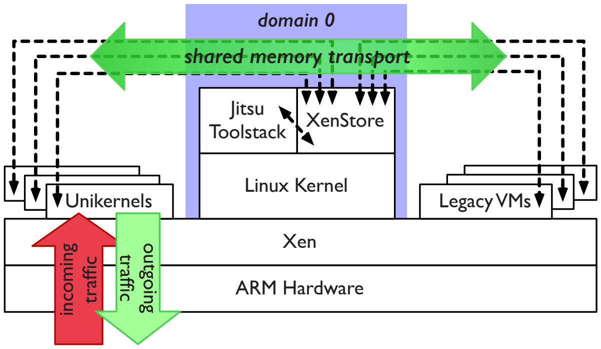


Figure 1: Jitsu architecture: external network connectivity is handled solely by memory-safe unikernels connected to general purpose VMs via shared memory.

2 Embedded Unikernels

Building software for embedded systems is typically more complex than for standard platforms. Embedded systems are often power-constrained, impose soft real-time constraints, and are designed around a monolithic firmware model that forces whole system upgrades rather than upgrade of constituent packages. To date, general-purpose hypervisors have not been able to meet these requirements, though microkernels have made inroads [9].

Several approaches to providing application isolation have received attention recently. As each provides different trade-offs between security and resource usage, we discuss them in turn (§2.1), motivating our choice of unikernels as our unit of deployment. We then outline the new Xen/ARM port that uses the latest ARM v7-A virtualization instructions (§2.2) and provide details of our implementation of a single-address space ARM unikernel using this new ABI (§2.3).

2.1 Application Containment

Strong isolation of multi-tenant applications is a requirement to support the distribution of application and system code. This requires both isolation at runtime as well as compact, lightweight distribution of code and associated state for booting. We next describe the spectrum of approaches meeting these goals, depicted in Figure 2.

OS Containers (Figure 2a). FreeBSD Jails [19] and Linux containers [38] both provide a lightweight mechanism to separate applications and their associated kernel policies. This is enforced via kernel support for isolated namespaces for files, processes, user accounts and other global configuration. Containers put the entire monolithic kernel in the trusted computing base, while still preventing applications from using certain functionality. Even the popular Docker container manager does not yet support isolation of root processes from each other.[1]

Both the total number and ongoing high rate of discovery of vulnerabilities indicate that stronger isolation is highly desirable (see Table 2). An effective way to achieve this is to build applications using a *library operating system* (libOS) [10, 24] to run over the smaller trusted computing base of a simple hypervisor. This has been explored in two modern strands of work.

Picoprocesses (Figure 2b). Drawbridge [34] demonstrated that the libOS approach can scale to running Windows applications with relatively low overhead (just 16MB of working set memory). Each application runs in its own *picoprocess* on top of a hypervisor, and this technique has since been extended to running POSIX applications as well [15]. Embassies [22] refactors the web client around this model such that untrusted applications can run on the user's computer in low-level native code containers that communicate externally via the network.

Unikernels (Figure 2c). Even more specialised applications can be built by leveraging modern programming languages to build *unikernels* [25]. Single-pass compilation of application logic, configuration files and device drivers results in output of a single-address-space VM where the standard compiler toolchain has eliminated unnecessary features. This approach is most beneficial for single-purpose appliances as opposed to more complex multi-tenant services (§5).

Unikernel frameworks are gaining traction for many domain-specific tasks including virtualizing network functions [29], eliminating I/O overheads [20], building distributed systems [6] and providing a minimal trust base to secure existing systems [11, 7]. In Jitsu we use the open-source MirageOS[2] written in OCaml, a statically type-safe language that has a low resource footprint and good native code compilers for both x86 and ARM. A particular advantage of using MirageOS when working with Xen is that all the toolstack libraries involved are written entirely in OCaml [36], making it easier to safely manage the flow of data through the system and to eliminate code that would otherwise add overhead [18].

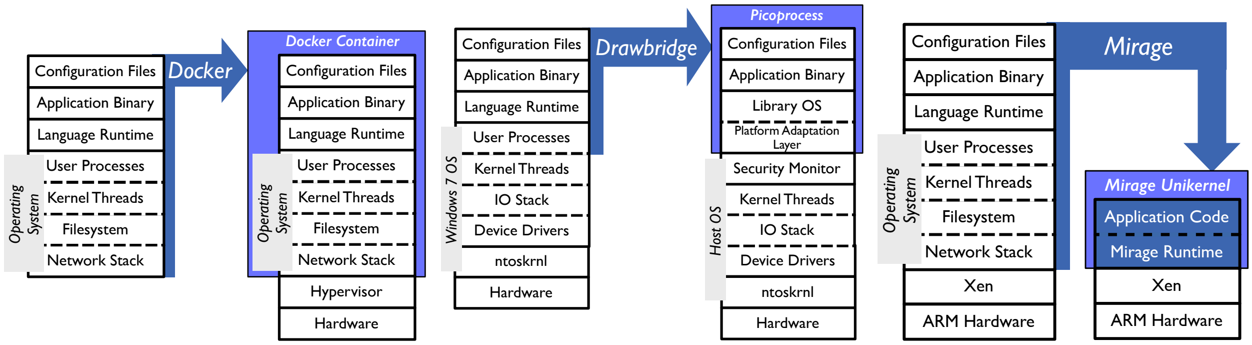
2.2 ARM Hardware Virtualization

Xen is a widely deployed type-1 hypervisor that isolates multiple VMs that share hardware resources. It was originally developed for x86 processors [2], on which it now provides three execution modes for VMs: paravirtualization (PV), where the guest operating system source is directly modified; hardware emulation (HVM), where specialised virtualization instructions and paging features available in modern x86 CPUs obviate the need to modify guest OS source code; and a hybrid model (PVH) that enables paravirtualized guests to use these newer hardware features for performance.[3]

[1] `https://docs.docker.com/articles/security/`

[2] `http://www.openmirage.org`

[3] See Belay et al [4] for an introduction to the newer VT-x features.



(a) Containers, e.g., Docker. (b) Picoprocesses, e.g., Drawbridge. (c) Unikernels, e.g., MirageOS.

Figure 2: Contrasting approaches to application containment.

The Xen 4.4 release added support for recent ARM architectures, specifically ARM v7-A and ARM v8-A. These include extensions that let a hypervisor manage hardware virtualized guests without the complexity of full paravirtualization. The Xen/ARM port is markedly simpler than x86 as it can avoid a range of legacy requirements: e.g., x86 VMs require qemu device emulation, which adds considerably to the trusted computing base [7]. Simultaneously, Xen/ARM is able to share a great deal of the mature Xen toolstack with Xen/x86, including the mechanics for specifying security policies and VM configurations.

Jitsu can thus target both Xen/ARM and Xen/x86, resulting in a consistent interface that spans a range of deployment environments, from conventional x86 server hosting environments to the more resource-constrained embedded environments with which we are particularly concerned, where ARM CPUs are commonplace.

2.3 Xen/ARM Unikernels

Bringing up MirageOS unikernels on ARM required detailed work mapping the libOS model onto the ARM architecture. We now describe booting MirageOS unikernels on ARM, their memory management requirements, and device virtualization support.

Xen Boot Library. The first generation of unikernels such as MirageOS [26, 25] (OCaml), HaLVM [11] (Haskell) and the GuestVM [32] (Java) were constructed by forking *Mini-OS*, a tiny Xen library kernel that initialises the CPU, displays console messages and allocates memory pages [39]. Over the years, Mini-OS has been directly incorporated into many other custom Xen operating systems, has had semi-POSIX compatibility bolted on, and has become part of the trusted computing base for some distributions [7]. This copying of code becomes a maintenance burden when integrating new features that get added to Mini-OS. Before porting to ARM, we therefore rearranged Mini-OS to be installed as a system library, suitable for static linking by any unikernel.[4] Functionality not required for booting was extracted into separate libraries, e.g., `libm` functionality is now provided by OpenLibM (which originates from FreeBSD's `libm`).

An important consequence of this is that a `libc` is no longer required for the core of MirageOS: *all* `libc` functionality is subsumed by pure OCaml libraries including networking, storage and unicode handling, with the exception of the rarely used floating point formatting code used by `printf`, for which we extracted code from the musl `libc`. Removing this functionality does not just benefit codesize: these embedded libraries are both security-critical (they run in the same address space as the type-safe unikernel code) and difficult to audit (they target a wide range of esoteric hardware platforms and thus require careful configuration of many compile-time options). Our refactoring thus significantly reduced the size of a unikernel's trusted computing base as well as improving portability.

Fast Booting on ARM. We then ported Mini-OS to boot against the new Xen ARM ABI. This domain building process is critical to reducing system latency, so we describe it here briefly. Xen/ARM kernels use the Linux zImage format to boot into a contiguous memory area. The Xen domain builder allocates a fresh virtual machine descriptor, assigns RAM to it and loads the kernel at the offset `0x8000` (32KB). Execution begins with the `r2` register pointing to a Flattened Device Tree (FDT). This is a similar key/value store to the one supplied by native ARM bootloaders and provides a unified tree for all further aspects of VM configuration. The FDT approach is much simpler than x86 booting, where the demands of supporting multiple modes (paravirtual, hardware-assisted and hybrids) result in configuration information being spread across virtualized BIOS, memory and Xen-specific interfaces.

[4] Our Mini-OS changes have been released back to Xen and are being integrated in the upstream distribution that will become Xen 4.6.

Some assembler code then performs basic boot tasks:

- Configuring the MMU, which handles mapping virtual to physical memory addresses.
- Turning on caching and branch prediction.
- Setting up the exception vector table, defining how to handle interrupts and deal with various faults such as reading from an invalid memory address.
- Setting up the stack pointer and jumping into the C `arch_init` function for the remainder of execution.

The early C code sets up the virtual logging console and interrupt controllers. After this, unikernel-specific C code binds interrupt handlers, memory allocators, timekeeping and grant tables [42] into the language runtime. The final step is to jump into the OCaml code section[5] and begin executing application logic. The application links memory-safe OCaml libraries to perform the remaining functions of device drivers and network stacks.

Modifying Memory Management. Once the MirageOS/ARM unikernel has booted, it runs in a single address space without context switching. However, the memory layout under ARM is significantly different from that for x86. Under the ARM Virtualization Extensions, there are two stages to converting a virtual memory address (used by application code) to a physical address in RAM, both of which go through translation tables. The first stage is under the control of the guest VM, where it maps the virtual address to what the guest believes is the physical address – the Intermediate Physical Address (IPA). The second stage, under the control of Xen, maps the IPA to the real physical address.

MirageOS' memory needs are very simple compared with traditional guest OSs. Most memory is provided directly to the managed OCaml heap which is grown on-demand. Unikernels will typically also allocate a few pages for interacting directly with Xen as these must be page-aligned and static, and so cannot be allocated on the garbage collected OCaml heap.

Although Xen does not commit to a specific fixed address for the IPA, the C code does need to run from a known location. To resolve this, the assembler boot code uses the program counter to detect where it is running and sets up a virtual-to-physical mapping that will make it appear at the expected location by adding a fixed offset to each virtual address. The table below shows this for Xen 4.5 (the latest stable release). The physical address is always at a fixed offset from the virtual address and addresses wrap around, so virtual address `0xC0400000` maps back to physical address `0` in this example.

Addresses		
Virtual	Physical	Purpose
`0x400000`	`0x40000000`	Stack (16 KB)
`0x404000`	`0x40004000`	Translation tables (16 KB)
`0x408000`	`0x40008000`	Kernel image

The stack, which grows downwards, is placed at the start of RAM so that an overflow will trigger a page fault that can be caught, and can also be grown in size later in the boot process when all of the RAM is available. The 16KB translation table is an array of 4-byte entries each mapping 1MB of the virtual address space, so the 16KB table is able to map the entire 32-bit address space (4GB). Each entry can either give the physical section address directly or point to a second-level table mapping individual 4KB pages; MirageOS implements the former as this reduces possible delays due to TLB misses.

The kernel code is followed by the `data` section containing constants and global variables, then the `bss` section with data that is initially zero and thus need not be stored in the kernel image, and finally the rest of the RAM under control of the memory allocator.

Device Virtualization. On Xen/x86 it is possible to add virtual devices by two means: pure PV devices that operate via a split-device model [42], and emulated hardware devices that use the `qemu` device emulator to provide the software model. Xen/ARM does not support the more complex hardware emulation at all, instead mandating (as a new ABI) that VMs support the Xen PV driver model to attach virtual devices.

MirageOS includes OCaml library implementations of the Xen PV protocols for networking and storage. The only modifications required from their x86 versions were the architecture-dependent memory barrier assembly instructions that differ between x86 and ARM, accessed via the OCaml foreign function interface.

The result of this work is to bring the benefits of MirageOS unikernels (compact, specialised appliances without excess baggage) to the resource-constrained ARM platform, providing an alternative to running full Linux or FreeBSD VMs. While we have described the specifics of the MirageOS port here, other teams have already picked up our work for their respective projects and are adapting it for other runtimes such as Click and Haskell.

3 The Jitsu Toolstack

We turn now to the Jitsu toolstack which supports the low-latency on-demand launching of the unikernels in response to network traffic. Our goal is to ensure that services listening on a network endpoint are always available to respond to traffic, but are otherwise not running to reduce resource utilisation. Jitsu is the Xen equivalent of the venerable `inetd` service on Unix, but instead of starting a process in response to incoming traffic, it starts a unikernel that can respond to requests on that IP address. While there have been wide-area versions of this approach in the past [1], we believe this is the first time it

[5] The `ocamlopt` compiler outputs standalone native code ARM object files that are linked with the garbage collector runtime library.

has been implemented with such low latency in a single embedded host without sacrificing isolation.

We describe Jitsu in three phases, each of which progressively reduces end-to-end latency. First, the traditional Xen toolstack is highly serialised across multiple blocking internal components, leading to large boot times due to long pauses between actual boot activity. We thus reduce these boot times by reducing this blocking behaviour and speeding up various boot components (§3.1). Jitsu preserves the existing boot protocol so that the many millions of existing Xen VM images will continue to work.

Second, we describe optimisation of the inter-VM communications protocol via *conduits*, a Plan9-like extension to support direct shared memory communication between named endpoints (§3.2). Conduits eliminate the need to use local networking to communicate between Jitsu and unikernels, further driving down latency.

Third, we introduce the *Synjitsu* directory service that masks boot latency to external clients by handling the initial stages of TCP handshake, only to hand-off the resulting state via a local conduit while the unikernel service completes booting and attaches to the network bridge (§3.3).

The net result is that a service VM can "cold boot" and respond to a TCP client in around 300–350ms, and an already-booted service can respond to local traffic in around 5ms (§4). In all cases, *all* network traffic is handled via memory-safe code in an unprivileged Xen VM.

3.1 Optimising Boot Times

Jitsu builds on the existing Xen toolstack by extending XenStore, a storage space shared between all VMs running on a physical host [12]. XenStore is a hierarchical, transactional key-value store where keys describe a path down a tree, and values store configuration and live status information for domains. Each running domain on a Xen instance has its own subtree, and so communication between domains can be coordinated via XenStore.

There are several stages to a VM booting that are triggered by XenStore: (*i*) a domain builder process loads the guest kernel image and configures it within a Xen datastructure before launching it; (*ii*) the new VM boots and attaches to its virtual devices, most notably a logging console and a network device; (*iii*) the remote end of the network and console device rings are attached to the backends that bridge them; and finally, (*iv*) the userspace starts and applications begin serving traffic.

Jitsu's utility relies on the ability to launch new VMs very quickly. Using the vanilla Xen toolstack, VM boot times are far too high for this, typically 3–5 seconds with high CPU usage for a Linux VM — hardly "just in time" when trying to start a network service with imperceptible client delay. Jitsu applies three optimisations to significantly reduce this, achieving lowest latency when booting a specialised unikernel instead of a generic VM.

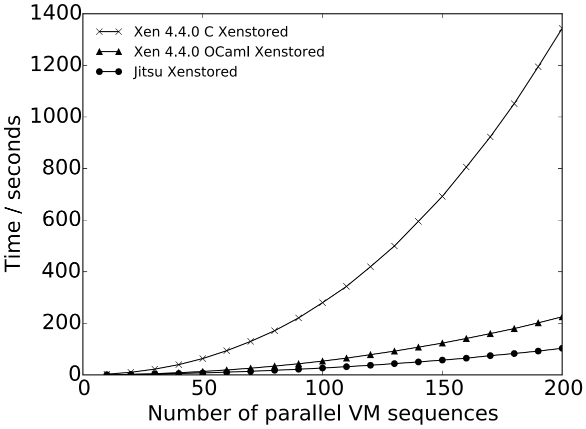


Figure 3: Comparison of different transaction reconciliation implementations during VM start/stop.

(*i*) **Domain building**. Xen's domain builder creates the initial VM kernel image. Most of its work is to initialise and zero out physical memory pages, thus guests with less memory are naturally built more quickly. As unikernels require such small amounts of memory to boot (8MB is plenty), they have an advantage over modern Linux distributions which typically require at least 64MB and are often recommended 128MB or more.

(*ii*) **Parallel device attachment**. While modern Linux parallelises much of its boot process, individual devices still have a serialisation overhead. The console device, for example, attaches to a dom0 `xenconsoled` service that drains the VM output and logs it. More significantly, attaching the network driver requires the backend domain to create a `vif` device in dom0, and to add it to a network bridge so that it can receive traffic. This blocks the VM while a slew of RPCs go back-and-forth between it and dom0, where hotplug shell scripts are executed.

This can be further sped up by parallelising the entire device attachment cycle with the domain builder itself. Jitsu starts the `vif` creation process before the domain builder runs, resulting in the two running in parallel. Although we could eliminate this overhead entirely by pre-creating domains and attaching them to the bridge (making VM launch simply a matter of attaching a unikernel to a domain before unpausing it), we prefer not to pay the cost of increased memory usage that would result from the pre-created domains.

(*iii*) **Transaction Deserialisation**. As the domain is built, a series of XenStore operations coordinates the multiple components involved in booting a VM. Building just one domain involves many transactional operations, and it becomes a latency bottleneck if they do not parallelise well. There are two XenStore implementations provided by upstream Xen: the default is a C implementation with filesystem-based transactions, and the other an alternative written in OCaml that uses in-memory transactions with merge functions that reduce the number of conflicts [12]. We further improved the OCaml XenStore transaction handling in a Jitsu-specific fork by providing a custom merge function that handles common directory roots in parallel transactions.

Figure 3 shows the dramatic differences in VM start time when doing VM start/stop operations in parallel, with the OCaml implementations clearly more efficient than the default dæmon in C. This is due to the reduced number of conflicts which otherwise cause the toolstack to cancel and retry a large set of domain building RPCs.

Figure 4 breaks down the impact of the domain creation optimisations. The test builds the VM image with a console and network interface and starts it. As this measures only VM construction time, not boot time, it applies to both unikernels and Linux VMs. Memory usage is a significant factor in domain creation, with a 256MB domain taking a full second to create, and a 16MB domain (suitable for a unikernel) still taking a significant 650ms. Rewriting the networking hotplug scripts to use the lightweight `dash` rather than the default `bash` reduces boot time to 300ms, and eliminating forking by invoking `ioctl` calls directly rather than running shell scripts further reduces boot time to 200ms. The final two optimisations to parallelise `vif` setup and asynchronously attach the console give the end result of 120ms to boot on ARM.

Jitsu is fully compatible with x86 as well as ARM, and so we ran the same tests on a 2.4GHz quad-core AMD x86_64 server to compare boot times against ARM. The most optimised VM creation time was just 20ms on x86 – around 6 times faster than the lower powered ARM board. Although we are focused on embedded deployments in this paper, it is worth noting that such fast boot times are possible in situations where power consumption is less of a concern (§4).

3.2 Communication Conduits

Coordinating a set of running unikernels requires some means to communicate between them. For conventional VMs, all such communication passes via shared memory rings to real hardware running in a privileged VM [42]. Device-specific RPC protocols are built over these rings to provide traditional abstractions such as `netfront` (network cards) or `blkfront` (mass storage).

This is a convenient abstraction when virtualizing existing OS kernels to run under Xen, as each protocol fits into the existing device driver framework. However, the lack of user/kernel space divide in a unikernel means that it links in device drivers as normal libraries: there is no need to fit the protocols into any existing abstraction. It becomes easy to construct custom RPC layers for communication between unikernels, whether instantiated as VMs on Xen or as Linux processes.

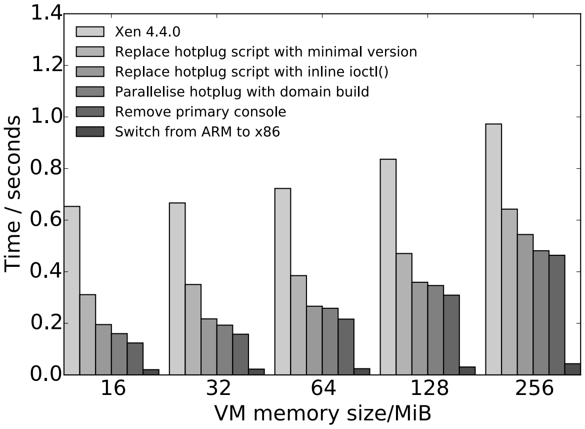


Figure 4: Optimising Xen/ARM domain build times.

Jitsu provides an abstraction over such a shared-memory communication protocol called *Conduit*, which (*i*) establishes shared-memory pages for zero-copy communication between peers; (*ii*) provides a rendezvous facility for VMs to discover named peers; and (*iii*) hooks into higher level name services like DNS. Conduit is designed to be compatible with the `vchan` library for inter-VM communication.[6]

3.2.1 Establishing a fast point-to-point connection

A `vchan` is a point-to-point link that uses Xen grant tables to map shared memory pages between two VMs, using Xen event channels to synchronise access to these pages. Establishing a `vchan` between two VMs requires each side to know its peer's domain id before the shared memory connection can be established. This allows `vchan` to work early in Xen's bootcycle before XenStore is available (e.g., within a disaggregated system [7]). Unlike previous inter-VM communication proposals [43], `vchan` remains simple by not mandating any rendezvous mechanism across VMs, focusing solely on providing a fast shared memory datapath.

Modern Linux kernels provide userspace access to grant mappings (`/dev/gntmap`) and event channels (`/dev/evtchn`), so we implemented the `vchan` protocol in pure OCaml using these devices. This required fixing several bugs in upstream Linux arising from the many ways to deadlock the system when interacting between user and kernel space. The lack of such a divide in unikernels made implementing this protocol for Mi-

[6] vchan was introduced by Qubes OS and later upstreamed to Xen; `http://openmirage.org/blog/introducing-vchan`

rageOS far simpler. The resulting code allows unikernel and Linux VMs on the same host to communicate without the overhead of a local network bridge.

3.2.2 Listening on named endpoints

For convenience, Conduit provides a higher-level rendezvous interface above `vchan` by using the existing XenStore metadata store. It extends the XenStore namespace in two places: the existing `/local/domain` tree for per-VM metadata, and a new `/conduit` tree for registering endpoint names and tracking established flows. Figure 5 shows a XenStore fragment with an HTTP client VM connecting to a HTTP server. When the server VM boots, it registers a name mapping from its domain id to `/conduit/http_server`. It then watches the `listen` key for any incoming connections.[7] The client VM similarly registers `/conduit/http_client` when it starts.

The `http_client` picks a unique port name and attempts to "resolve" the `http_server` target by writing the port name to the `listen` queue for the server (e.g., `/conduit/http_server/listen/conn1`). The server VM receives a watch event and reads the remote domain id and port name from its listen queue, giving it sufficient information to establish a `vchan`. The connection metadata is written into `/local/domain/<domid>/vchan`, and contains the grant table and event channel references through which both sides obtain their shared memory pages and virtual interrupts. The server also updates the `/flows` table with extra metadata such as per-flow statistics that can be read by management tools.

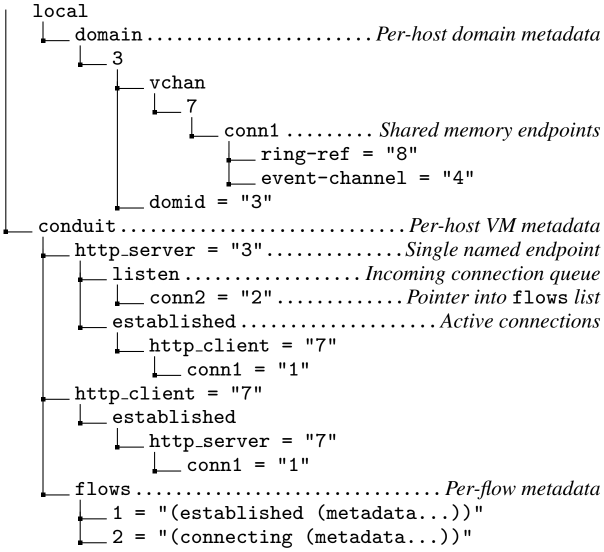


Figure 5: The XenStore tree layout for coordinating the establishment of inter-VM shared memory channels.

3.2.3 Access control and transactions

XenStore already has an access control model that allows per-domain access control over keys and their child nodes. This is a good fit for Conduit except during initial setup where the client domain must write directly into the `listen` directory published by the server. Although the directory is open for writing from any other VM, new keys must be restricted to only be readable by the directory owner and the creator of the key. This is analogous to setting the `setgid` and sticky bits in POSIX filesystems. With this extension added to XenStore, domains cannot observe or interfere with the creation of conduits that do not concern them, and only XenStore itself is required for rendezvous.

As XenStore is already a filesystem-like interface, this protocol is similar to the Plan 9 network model [33], with a few notable differences: (*i*) although connection establishment goes through XenStore, established channels are zero-copy shared memory endpoints that no longer require any interaction with XenStore; and (*ii*) XenStore provides a transactional interface to let batch updates be committed atomically [12]. This eliminates potential inconsistencies arising from having state metadata spread over several keys (such as `/conduit/flows/1` and `/local/domain/3/vchan` in Figure 5).

The Conduit interface enables us to write unikernel code without having to know in advance where the remote peer is running. In the example above, the `http_server` might be a Xen unikernel or a normal Linux guest VM listening from a userspace Unix binary. Unikernels also need not trust each other as they act as a distributed system on a single host [3], communicating via a bytestream rather than directly sharing pointers into each other's address spaces.[8]

3.3 The Jitsu Directory Service

Our goal is to ensure that unikernels are launched and halted in real-time in response to network requests. This role is similar to that performed by `inetd` on Unix, and is fulfilled by the Jitsu Directory Service that maps external DNS [31] requests onto unikernel instances. When the unikernel for a service has launched, it can serve as many requests as a single VM can handle – we typically launch a VM per registered service, not one per TCP connection.

A Jitsu VM is launched at boot time with access to the external network and handles name resolution, invoked either by a local unikernel over a conduit, or through DNS protocol handlers listening on the network bridge. In the former case, the Jitsu resolver is discovered via a well-known `jitsud` Conduit node, while in the latter it

[7] A *watch* is the XenStore term for registering for notification callbacks whenever any key or value in a watched subtree is modified.

[8] The J-Kernel [40] and FlowCaml [37] provide a guide as to how pointer sharing could be safely built into future revisions of MirageOS.

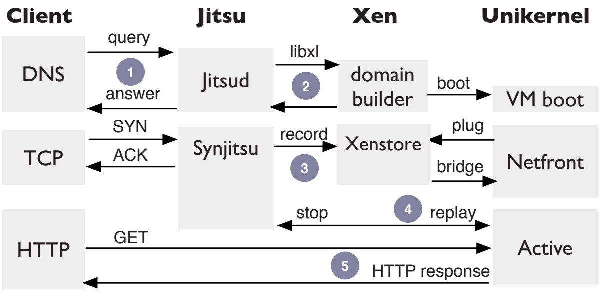


Figure 6: How Jitsu masks boot latency. ❶ DNS request triggers unikernel launch; ❷ response sent when domain building completes but before networking is active; ❸ TCP requests are buffered into XenStore until bridging is setup; and ❹ the active unikernel replays the buffered connections before ❺ directly serving traffic.

is discovered through the usual process in DNS (e.g., resolving `ns.domain.name`). If a name resolution request is received that maps onto a running unikernel, Jitsu just returns an appropriate IP address or `vchan` endpoint.

If the name requested does *not* correspond to a running unikernel, Jitsu launches the desired unikernel while simultaneously returning an appropriate endpoint (again, IP address or `vchan`) against which the client can start the higher level protocol interaction (e.g., a TCP three-way handshake). However, while the VM is starting it will not be ready to respond to network traffic as the network bridging subsystem connects asynchronously. This opens a race condition where the DNS response has been sent to the client, but the unikernel is not yet listening for the TCP `SYN` packet that will follow (likely very quickly as the client is typically local). The `SYN` packet is dropped, and the client retransmits after 1s – well outside our low-latency requirement.

3.3.1 Connection proxying via Synjitsu

We could remove this race condition by delaying the initial DNS response until the unikernel network is fully established. Instead we take advantage of the high-level libOS network stack available to us to provide a lower latency solution: we explicitly handle incoming connections in a proxy unikernel, and hand off the state to the full unikernel once it has finished plugging its network device in. This helps Jitsu to mask any latency associated with booting the target unikernel, as well as making it more robust in the face of TCP connections arriving unexpectedly outside of DNS resolution (e.g., because a client did not respect the TTL in a DNS response and attempted to connect to the service directly).

Figure 6 shows the packet flow with the `synjitsu` unikernel performing this connection proxying. When a DNS request comes in, the unikernel boot process starts, returning a DNS response as soon as the VM resource allocation is complete (resource exhaustion can thus be returned in the DNS response as a `SERVFAIL` to indicate the client should go elsewhere). As unikernel boot (20ms on x86, 350ms on ARM) takes longer than the RTT of a packet on a local network (5ms), it is likely that a TCP SYN would follow and be lost before the unikernel has booted, triggering a slow TCP client retransmission.

```
conduit
└── http_server = "3"
    └── tcpv4 .................. Connection state from proxy
        ├── 1 ................ Received SYN but not responded
        │   ├── state = "SYN"
        │   └── tcb = "(src ...)(port ...)"
        └── 2 .............. Established and buffering packets
            ├── state = "SYN_ACK"
            ├── tcb = "((port ...)(isn ...))"
            └── packets = "((data ...)(...))"
```

Figure 7: The `synjitsu` proxy registers embryonic TCP connections to mask unikernel startup time.

`synjitsu`, built using the same OCaml TCP stack as the booting unikernel, removes this race entirely by listening on the external network bridge and an internal conduit for TCP packets destined for a unikernel that is still booting. When it receives a `SYN`, it writes entries into a special area in the conduit XenStore tree for the booting unikernel. Figure 7 shows two examples; (*i*) where a `SYN` has been received but not responded to, and (*ii*) where a `SYN_ACK` has been sent by the proxy and the TCP data stream buffered up. When the unikernel finishes booting and has an active network interface, it signals to `synjitsu` that it is ready for traffic via a two-phase commit in XenStore, ensuring only one of them ever handles any given packet. The unikernel then reconstructs the TCP state descriptors based on the recorded state, and handles subsequent traffic on the bridge directly, with no further interference from `synjitsu`.

Splitting state across a dormant kernel and a proxy is not a new technique [1], but the high-level nature of the OCaml TCP/IP stack makes implementation a simple matter of (de)serialising values across XenStore. As only one of `synjitsu` or the unikernel ever replies to a packet, we avoid the complexity and latency increase from building a distributed network stack [16] within the host. It is also relatively easy to extend to higher-level protocols such as SSL/TLS [30], e.g., to perform the 7-way initial key exchange in one VM before it hands off the connection to another unikernel that has no access to the private keys for the remainder of its lifetime.

3.3.2 Service Configuration

Consider a client wishing to access one of a set of low-traffic websites, such as a set of personal homepages and photographs. Hosting each of these relatively low traffic

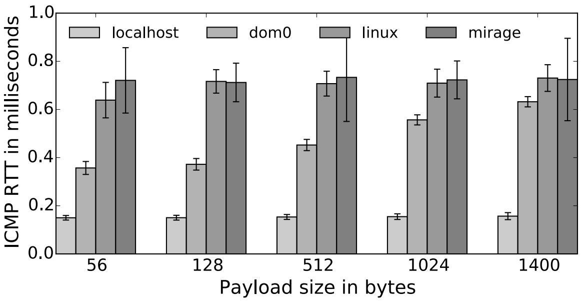


Figure 8: ICMP RTT showing the datapath latency.

Power Usage (W, 5V)		Board Model
Idle	Spinning	and active components
1.43	2.61	Cubieboard2
2.10	2.58	Cubieboard2 +Ethernet
3.36	4.49	Cubieboard2 +SSD
4.06	4.51	Cubieboard2 +SSD+Ethernet
1.72	2.86	Cubietruck
2.58	3.76	Cubietruck +Ethernet
3.92	5.51	Cubietruck +SSD
4.91	6.26	Cubietruck +SSD+Ethernet
6.84	27.02	*Intel Haswell NUC [35]*

Table 1: Power usage of the ARM boards when running Xen, with reported Intel results for comparison.

sites in the cloud would be a waste of money, while a typical small home router or similar is unlikely to have sufficient resources to keep them all simultaneously live yet isolated. An ARM device using the Jitsu toolstack is registered in the public DNS as `ns.family.name`, the nameserver for the family.name zone. When a DNS request comes in for `alice.family.name`, Jitsu returns the local external IP address configured for Alice's unikernel and performs connection proxying while Alice's unikernel launches. Conventional failover models are supported – multiple ARM boards could be registered in the DNS and return `SERVFAIL` responses if they do not have resources to serve the traffic.

In our current implementation, the Jitsu services are statically configured via OCaml code to map their unikernel with an IP address, protocol and port. We expose publication of running services via the DNS, as either an authoritative server or recursive resolver. More dynamic configurations, where launched unikernels may themselves alter the name–address–unikernel mappings and can publish using, e.g., Dynamic DNS are possible to build over this lower-level interface.

4 Evaluation

Our tests are conducted on two inexpensive off-the-shelf ARM boards: a Cubieboard2 (dual-core Allwinner ARM A20, 1GB RAM, 100Mb Ethernet) and a Cubietruck (same CPU, 2GB RAM, 1Gb Ethernet) running Xen 4.4 and an Ubuntu 14.04 dom0.[9] Our evaluation aims to answer the following questions:

- Does the port to the ARM architecture have reasonable performance, latency and energy efficiency?
- Is launching services in isolated Xen VMs a viable alternative to other approaches, e.g., containers?
- Is there any benefit in the extra isolation afforded by a type-1 hypervisor?

Throughput. We have previously carried out a fuller analysis of unikernel throughput with various protocols [25], so here we are simply verifying that there is no regression on ARM. As the ARM CPUs are considerably underpowered compared to x86 CPUs, we built a HTTP persistent queue service in MirageOS to ensure that network throughput remains acceptable. The working set of this service is larger than available RAM, and so it is served from disk. After some optimisations,[10] it served HTTP traffic at a rate of 57.92Mb/s, at which point it becomes disk bound. An `iperf` test with checksum offloading enabled revealed the same performance for Linux and MirageOS VMs.

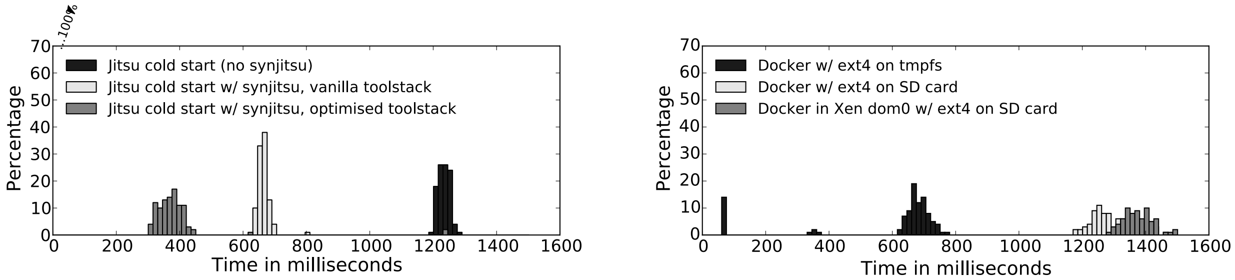
Datapath latency. The imposition of Xen and type-safety risks introducing additional latency in the network datapath, and so Jitsu minimises excess bridging (§3.2) and proxying on the data plane (§3.3.1). Figure 8 plots ICMP latency when pinging the client's own external interface (i.e., the latency of the client stack), the Xen dom0, a Linux Xen/ARM VM and a type-safe MirageOS unikernel VM. The latency difference between a Linux and MirageOS VM is never more than 0.4ms, although MirageOS does have slightly more variation.

Service Startup Latency. Figure 9a shows the end-to-end latency of HTTP requests from an external network client. First we measure the time for a "cold start" when no unikernel was running and so one had to be started by Jitsu. Early `SYN` packets are lost and the client (running Linux) retransmits them, leading to response times of over a second. We then show the effects of running `synjitsu` to proxy connection setup by intercepting `SYN` packets and handing them over to the unikernel, and also the effect of the toolstack optimisations to improve VM creation time (§3.1). Finally the latency for an already-running service is imperceptible as expected. We do not plot the start time of a full Ubuntu Linux VM, since it took over 5s with the default distribution image.

We also tested Docker 1.2.0 Linux container startup triggered from `inetd` to compare its latency with VMs. A container's start latency on a Cubieboard2 is dominated by disk I/O (Figure 9b). When running directly from a 10MB/s SD card, Docker takes at least 1.1s (na-

[9] SD card images are found at `http://blobs.openmirage.org`

[10] For full details on the profiling, see `http://bit.ly/Y3kuun`



(a) Using Jitsu from a cold start (booting a new unikernel) without Synjitsu, with Synjitsu and the ordinary toolstack, and with Synjitsu and the optimised toolstack.

(b) Using Docker: direct on Linux with RAM filesystem (`tmpfs`) and SD card, and on Xen in dom0.

Figure 9: HTTP request response times for Jitsu and Docker.

tive Linux) or 1.2s (under Xen) to spawn a new container in response to a request. To understand the effect of slow storage on Docker's start time, we also mounted Docker's volumes on an `ext4` loopback volume inside of a `tmpfs`.[11] In this configuration, container start times remained at 600ms or higher, considerably higher than Jitsu unikernels. This configuration also generated buffer IO, `ext4` and VFS errors in a significant fraction of tests resulting in early process termination.

Power Usage. A key facet of our contribution is that by using ARM-based devices, power consumption is significantly reduced, to the extent that they become acceptable to run 24/7 in a domestic environment. Table 1 shows the power usage of various configurations of our evaluation boards, measured using a custom power measurement unit we built that intercepts the USB power link to the boards. We measured each board when idle (just running Xen and a dom0), spinning in a busy loop and with Ethernet and an external solid-state drive. The SSD almost doubled power usage, but the small binary size of unikernels (around 1MB) means that in many cases we do not require a lot of space beyond that provided by the internal MMC flash. We failed to find an equivalent Intel board in the same price/performance/functionality range as the Cubieboard, and so we report power usage on the Intel Haswell NUC [35]. We also powered a Cubieboard with a USB battery unit that ran for 9 hours while logging the date every minute.

Security. To evaluate the end-to-end security properties of the Jitsu design *vs* a more conventional Linux embedded system, we looked for critical security bugs eliminated by use of (*i*) isolation via a type-1 hypervisor and (*ii*) a memory-safe language to build minimal VM appliances. Table 2 compares a recent representative selection of CVE vulnerabilities against embedded network devices (top), the Linux kernel (middle), and Xen on ARM (bottom). With Jitsu, the top group would be entirely eliminated and the middle group largely eliminated, while the bottom group would remain.

The commonest vulnerabilities still arise from protocol parsers written in unsafe languages, resulting in remote code execution vulnerabilities across the spectrum of almost every common protocol found on edge routers. Jitsu ensures that *all* traffic parsed on the external network be done so in memory-safe OCaml, mitigating this class of overflows. Another recent non-buffer-overflow vulnerability of note is *ShellShock*, a recent parsing error in the `bash` shell (CVE-2014-6271) that permits remote code execution by manipulating environment variables. The unikernel design does not include a shell, and our latency optimisations in Jitsu (§3.1) also eliminate shell scripts from the security-critical management toolstack.

The middle stream of vulnerabilities that affect the Linux kernel motivate the use of a type-1 hypervisor like Xen rather than Linux containers. Only a few bugs that affect physical device drivers can harm Xen, and even those can be mitigated in future revisions of Jitsu via driver domains [7]. The bottom stream of vulnerabilities show the class of errors that have affected Xen/ARM since its first release, and none of these are exploitable remotely. Many of these are a result of the relatively immature Xen/ARM port which has seen just one public release to date. The simplicity of the Xen/ARM codebase compared to x86 may lend itself to formal specification and verification in the future [21].

More broadly, by enabling strong isolation *inside* embedded devices, new distributed system designs leveraging multi-tenancy and low latency are possible. Systems designed to take advantage of Jitsu's isolation properties protect themselves from many passive and active attacks on wide-area network links by transmitting less data over those links and using them only for hardened, general-purpose software distribution.

[11] This rather complex configuration was required as the device-mapper in Linux 3.16 does not work directly over `tmpfs` mounts

	CVE	Description	App	Remote	Execute	DoS	Exposure	Jitsu
Embedded systems	CVE-2011-3992	SSH overflow	✓	✓	✓	✓	✓	
	CVE-2012-1800	DCP overflow	✓	✓	✓	✓	✓	
	CVE-2013-0659	UDP overflow	✓	✓	✓	✓	✓	
	CVE-2013-1605	HTTP overflow	✓	✓	✓	✓	✓	
	CVE-2013-2338	SSO overflow	✓	✓	✓	✓	✓	
	CVE-2013-4977	RTSP overflow	✓	✓	✓	✓	✓	
	CVE-2013-4980	RTSP overflow	✓	✓	✓	✓	✓	
	CVE-2013-6343	HTTP overflow	✓	✓	✓	✓	✓	
	CVE-2014-0355	HTTP overflow	✓	✓	✓	✓	✓	
	CVE-2014-3936	HNAP overflow	✓	✓	✓	✓	✓	
Linux	CVE-2014-0077	KVM overflow			✓	✓	✓	
	CVE-2014-0100	IP fragmentation		✓		✓		
	CVE-2014-0155	KVM IOAPIC				✓		
	CVE-2014-0206	AIO kernel mem					✓	
	CVE-2014-1690	IRC netfilter	✓	✓			✓	
	CVE-2014-2309	IPv6 routing mem		✓		✓		
	CVE-2014-2672	Atheros WLAN DoS		✓		✓		✓
	CVE-2014-2706	MAC 802.11 race		✓		✓		✓
	CVE-2014-5206	MNT_NS bypass					✓	
	CVE-2014-5207	MNT_NS remount				✓	✓	
Xen	CVE-2014-2580	Net disable mutex				✓		✓
	CVE-2014-2915	Processor control				✓		✓
	CVE-2014-2986	NULL deref in VGIC				✓		✓
	CVE-2014-3125	Timer context switch				✓		✓
	CVE-2014-3714	Kernel load overflow				✓	✓	✓
	CVE-2014-3715	DTB append				✓	✓	✓
	CVE-2014-3716	DTB alignment				✓		✓
	CVE-2014-3717	Kernel load overflow				✓	✓	✓
	CVE-2014-3969	Vmem privs			✓	✓	✓	✓
	CVE-2014-4021	Dirty recovery					✓	✓
	CVE-2014-4022	Dirty init					✓	✓
	CVE-2014-5147	32-bit traps				✓		✓

Table 2: A representative selection of vulnerabilities in three key system components. **App** indicates an application vulnerability. **Remote** indicates remote exploitation potential. **Execute** indicates arbitrary code execution. **DoS** indicates denial of service potential. **Exposure** indicates data exfiltration potential. **Jitsu** indicates vulnerabilities that could affect a Jitsu system (Xen on ARM with a Linux Dom0 for network drivers).

5 Discussion

Jitsu solves the problems of supporting low-latency deployment of code requiring strong isolation to resource-constrained embedded platforms. Although we focus on use of MirageOS unikernels in that specific problem domain, the techniques embodied within Jitsu have a number of attendant benefits which we discuss here.

General Jitsu. Although we have focused in this paper on using Jitsu with unikernels, we have not made any changes to the Xen guest ABI. As a result Jitsu works as described with legacy VMs (e.g., Linux, FreeBSD) on both ARM and in traditional x86 datacenter environments. This contrasts with systems such as ClickOS [28] which modifies the ABI to achieve very dense, highly parallel deployments of 10,000s of VMs in an x86_64 datacenter. We anticipate that both of these approaches will converge in upstream Xen in the future through a revision of the XenStore protocol. The one thing that Jitsu cannot provide with legacy VMs is guaranteed latency, due to the inherent boot overheads of such VMs. Tests on x86 (Figure 4) point to the intriguing possibility of very fast 20–30ms response times in datacenter environments as well.

Jitsu can easily be extended to support other VM lifecycle operations such as live relocation or VM forking [41, 23] in response to network requests. However, Jitsu is particularly well-suited to Xen/ARM through its use of explicit state transfer in `synjitsu` rather than depending on these hypervisor-level features. Forking or migrating entire VMs is more resource intensive than protocol state transfer, and is not yet fully supported by Xen/ARM 4.5. Simple TCP connection handover as in `synjitsu` is also easily extensible, and we are currently applying it to a full seven packet SSL/TLS handshake to support encrypted connections [30].

Finally, as noted previously, use of the Conduit stack for coordinating communication between VMs is not limited to unikernels. The basic principle of providing a name-based resolver to shared memory endpoints that does not depend on either a network (e.g., TCP/IP) or a process model (e.g., SysV `shmem`) can be used to interface conventional VM software stacks with unikernels.

Modularity. Jitsu both exploits and enables extensive use of modularity, which is very useful in building reliable distributed systems. The first form of modularity is found the way MirageOS is implemented – a set of lightweight OCaml libraries fulfilling module type signatures. Type-checking these signatures makes it easy to ensure that, when picking and choosing the features to be included in a particular unikernel, the basic system requirements are satisfied. As a result, developing features such as Conduit (§3.2) was far more straightforward than would have been for a traditional OS: during development it never crashed the Mini-OS kernel, and almost every error was caught and turned into an explicit condition or a high-level OCaml exception. Similarly, the `synjitsu` proxy uses the same OCaml TCP/IP library as found in the unikernels, simply with very different runtime policies.

The new Conduit capability also directly addresses one of the key criticisms of the MirageOS approach: lack of multilingual support through the dependence on OCaml. With Jitsu – specifically the combination of Synjitsu and Conduit's low latency high throughput inter-VM communication – it is entirely feasible to launch a TCP/IP MirageOS unikernel that will proxy incoming traffic to another unikernel (e.g., in Ruby or PHP) that need only implement the Conduit protocol and so need not expose, or even include, a TCP/IP implementation.

Use cases. We envisage Jitsu being useful in a wide range of situations. For example, where legacy software that may be difficult to upgrade (e.g., embedded device firmware) must be run, Jitsu can be used to provide a very narrow, application specific firewall that can filter and groom incoming traffic from the public Internet limiting the exposure of the legacy software.

Another useful scenario would be to contain application code that would normally run as a cloud service so that it can be run on a platform, such as the home router, inside the home. For example, consider the latency-sensitive applications noted earlier, Google Glass and Apple's Siri. By implementing the cloud services that support these applications as unikernels, they could be downloaded to run locally on the home router, providing significantly lower latency for common operations while still having the full power of the cloud at their disposal.

Yet other application scenarios include those where the data to be processed by the cloud-hosted service might be considered particularly personal, such as a family's photos. Photos might be hosted encrypted on the home router, and then unikernel versions of services such as Apple's iPhoto and Google's Picasa might be instantiated on-demand on the home router and given access to decryption keys held locally. Access to photos is then more directly controlled within the home without giving up all the personal data to the cloud providers [14].

Experimental artefacts. Finally, we wish to encourage use of Jitsu by other groups to explore the possibilities inherent in the platform. To that end we have made available all the code used in this paper:

- MirageOS and Jitsu are hosted on GitHub (`github.com/mirage`) with documentation on our self-hosted website at `openmirage.org`.
- The Xen/ARM and dom0 Linux distributions can be built via scripts at `github.com/mirage/xen-arm-builder`; and prebuilt SD Card images are also available for download there.
- Enquiries can be directed to our Xen.org mailing list linked from `openmirage.org/about/`.

6 Conclusions

We have presented *Jitsu*, a low latency toolstack for Xen/ARM that uses memory-safe unikernels to serve applications with significantly greater levels of isolation and security than currently achieved on modern embedded devices. Jitsu includes optimisations of the toolstack of Xen, a full-featured widely-deployed modern hypervisor that now supports ARM devices, maintaining full ABI compatibility for existing deployments. Jitsu adds the convenience of an `inetd`-like service that leverages our reduced boot latencies of around 350ms on ARM and 30ms on x86 to summon VMs in response to network traffic. Our *Synjitsu* service masks even that latency by minimally proxying connection setup requests to enable instantaneous response for clients while the unikernel boots.

The full source code is available under a BSD license at `openmirage.org`, along with documentation and installation instructions for use with Cubieboards. We welcome any patches, success stories and reports of improbable stunts conducted using Jitsu.

7 Acknowledgements

We thank Mark Shinwell, Hannes Mehnert, Raphael Proust, Malte Schwarzkopf, Matt Grosvenor, Ionel Gog, Hajime Tazaki and Martin Lucina, our paper shepherd Geoff Voelker and the anonymous NSDI reviewers for feedback. The research leading to these results received funding from the European Union's Seventh Framework Programme FP7/2007–2013 under the Trilogy 2 project (grant agreement no. 317756), and the User Centric Networking project, (grant agreement no. 611001), and the Defense Advanced Research Projects Agency (DARPA) and the Air Force Research Laboratory (AFRL), under contract FA8750-11-C-0249. The views, opinions, and/or findings contained in this report are those of the authors and should not be interpreted as representing the official views or policies, either expressed or implied, of the Defense Advanced Research Projects Agency or the Department of Defense.

References

[1] AGARWAL, Y., SAVAGE, S., AND GUPTA, R. Sleepserver: A software-only approach for reducing the energy consumption of PCs within enterprise environments. In *Proc. USENIX Annual Technical Conference (ATC)* (Boston, MA, 2010), USENIX Association, pp. 22–22.

[2] 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. 19th ACM Symposium on Operating Systems Principles (SOSP)* (Bolton Landing, NY, USA, 2003), ACM, pp. 164–177.

[3] BAUMANN, A., BARHAM, P., DAGAND, P., HARRIS, T., ISAACS, R., PETER, S., ROSCOE, T., SCHÜPBACH, A., AND SINGHANIA, A. The multikernel: a new OS architecture for scalable multicore systems. In *Proc. 22nd ACM Symposium on Operating Systems Principles (SOSP)* (Big Sky, MT, USA, Oct. 11–14 2009), pp. 29–44.

[4] BELAY, A., BITTAU, A., MASHTIZADEH, A., TEREI, D., MAZIÈRES, D., AND KOZYRAKIS, C. Dune: Safe user-level access to privileged CPU features. In *Proc. 10th USENIX Symposium on Operating Systems Design and Implementation (OSDI)* (Hollywood, CA, 2012), USENIX, pp. 335–348.

[5] CHECKOWAY, S., MCCOY, D., KANTOR, B., ANDERSON, D., SHACHAM, H., SAVAGE, S., KOSCHER, K., CZESKIS, A., ROESNER, F., AND KOHNO, T. Comprehensive experimental analyses of automotive attack surfaces. In *Proc. 20th USENIX Security Symposium (SEC)* (San Francisco, CA, 2011), USENIX Association, pp. 6–6.

[6] CLOUDOZER INC. Erlang on Xen, at the heart of super-elastic clouds. `http://erlangonxen.org/`.

[7] COLP, P., NANAVATI, M., ZHU, J., AIELLO, W., COKER, G., DEEGAN, T., LOSCOCCO, P., AND WARFIELD, A. Breaking up is hard to do: security and functionality in a commodity hypervisor. In *Proc. 23rd ACM Symposium on Operating Systems Principles (SOSP)* (Cascais, Portugal, Oct. 23–26 2011), pp. 189–202.

[8] COSTIN, A., ZADDACH, J., FRANCILLON, A., AND BALZAROTTI, D. A large-scale analysis of the security of embedded firmwares. In *Proc. 23rd USENIX Security Symposium (SEC)* (San Diego, CA, Aug. 2014), USENIX Association, pp. 95–110.

[9] ELPHINSTONE, K., AND HEISER, G. From L3 to seL4 what have we learnt in 20 years of L4 microkernels? In *Proc. 24th ACM Symposium on Operating Systems Principles (SOSP)* (Farminton, Pennsylvania, 2013), ACM, pp. 133–150.

[10] ENGLER, D. R., KAASHOEK, M. F., AND O'TOOLE, JR., J. Exokernel: an operating system architecture for application-level resource management. In *Proc. 15th ACM Symposium on Operating Systems Principles (SOSP)* (Copper Mountain, Colorado, USA, 1995), ACM, pp. 251–266.

[11] GALOIS INC. The Haskell Lightweight Virtual Machine (HALVM) source archive. `https://github.com/GaloisInc/HaLVM`.

[12] GAZAGNAIRE, T., AND HANQUEZ, V. Oxenstored: an efficient hierarchical and transactional database using functional programming with reference cell comparisons. *SIGPLAN Notices 44*, 9 (Aug. 2009), 203–214.

[13] GRANZER, W., PRAUS, F., AND KASTNER, W. Security in building automation systems. *IEEE Trans. Industrial Electronics 57*, 11 (Nov. 2010), 3622–3630.

[14] HADDADI, H., HOWARD, H., CHAUDHRY, A., CROWCROFT, J., MADHAVAPEDDY, A., AND MORTIER, R. Personal data: Thinking inside the box. Tech. Rep. abs/1501.04737, arXiv, Jan. 2015.

[15] HOWELL, J., PARNO, B., AND DOUCEUR, J. R. How to run POSIX apps in a minimal picoprocess. In *Proc. USENIX Annual Technical Conference (ATC)* (June 2013), USENIX.

[16] HRUBY, T., VOGT, D., BOS, H., AND TANENBAUM, A. S. Keep net working— on a dependable and fast networking stack. In *Proc. Dependable Systems and Networks (DSN)* (Boston, MA, June 2012).

[17] HYMAN, P. Augmented-reality glasses bring cloud security into sharp focus. *Commun. ACM 56*, 6 (June 2013), 18–20.

[18] JEONG, S., LEE, K., LEE, S., SON, S., AND WON, Y. I/O stack optimization for smartphones. In *Proc. USENIX Annual Technical Conference (ATC)* (San Jose, CA, 2013), USENIX, pp. 309–320.

[19] KAMP, P.-H., AND WATSON, R. N. M. Jails: Confining the omnipotent root. In *Proc. SANE Conference* (2000).

[20] Kivity, A., Laor, D., Costa, G., Enberg, P., Har'El, N., Marti, D., and Zolotarov, V. OSv—optimizing the operating system for virtual machines. In *Proc. USENIX Annual Technical Conference (ATC)* (Philadelphia, PA, June 2014), USENIX Association, pp. 61–72.

[21] Klein, G., Elphinstone, K., Heiser, G., Andronick, J., Cock, D., Derrin, P., Elkaduwe, D., Engelhardt, K., Kolanski, R., Norrish, M., Sewell, T., Tuch, H., and Winwood, S. seL4: formal verification of an OS kernel. In *Proc. 22nd ACM Symposium on Operating Systems Principles (SOSP)* (Big Sky, MT, USA, Oct. 11–14 2009), pp. 207–220.

[22] Krsul, I., Ganguly, A., Zhang, J., Fortes, J., and Figueiredo, R. VMPlants: Providing and managing virtual machine execution environments for grid computing. In *Proc. ACM/IEEE Supercomputing Conference (SC)* (Nov. 2004), p. 7.

[23] Lagar-Cavilla, H. A., Whitney, J. A., Scannell, A. M., Patchin, P., Rumble, S. M., de Lara, E., Brudno, M., and Satyanarayanan, M. SnowFlock: rapid virtual machine cloning for cloud computing. In *Proc. 4th ACM European Conference on Computer Systems (EuroSys)* (Nuremberg, Germany, 2009), pp. 1–12.

[24] Leslie, I. M., McAuley, D., Black, R., Roscoe, T., Barham, P. T., Evers, D., Fairbairns, R., and Hyden, E. The design and implementation of an operating system to support distributed multimedia applications. *IEEE Journal of Selected Areas in Communications 14*, 7 (1996), 1280–1297.

[25] Madhavapeddy, A., Mortier, R., Rotsos, C., Scott, D., Singh, B., Gazagnaire, T., Smith, S., Hand, S., and Crowcroft, J. Unikernels: library operating systems for the cloud. In *Proc. 18th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)* (Houston, Texas, USA, 2013), ACM, pp. 461–472.

[26] Madhavapeddy, A., Mortier, R., Sohan, R., Gazagnaire, T., Hand, S., Deegan, T., McAuley, D., and Crowcroft, J. Turning down the LAMP: Software specialisation for the cloud. In *Proc. 2nd USENIX Workshop on Hot Topics in Cloud Computing (HotCloud)* (Boston, MA, USA, June 2010).

[27] Madhavapeddy, A., and Scott, D. Unikernels: The rise of the virtual library operating system. *Communications of the ACM (CACM) 57*, 1 (Jan. 2014), 61–69.

[28] Manco, F., Martins, J., and Huici, F. Towards the super fluid cloud. In *Proc. ACM SIGCOMM (demo track)* (Chicago, Illinois, USA, 2014), ACM, pp. 355–356.

[29] Martins, J., Ahmed, M., Raiciu, C., Olteanu, V., Honda, M., Bifulco, R., and Huici, F. ClickOS and the art of network function virtualization. In *Proc. 11th USENIX Symposium on Networked Systems Design and Implementation (NSDI)* (Seattle, WA, Apr. 2014), USENIX Association, pp. 459–473.

[30] Mehnert, H., and Mersinjak, D. K. Transport Layer Security purely in OCaml. In *Proc. ACM OCaml 2014 Workshop* (Sept. 2014).

[31] Mockapetris, P. Domain names – implementation and specification. RFC 1035 (Standard), Nov. 1987. Updated by RFCs 1101, 1183, 1348, 1876, 1982, 1995, 1996, 2065, 2136, 2181, 2137, 2308, 2535, 2845, 3425, 3658, 4033, 4034, 4035, 4343, 5936, 5966, 6604.

[32] Oracle. GuestVM. `http://labs.oracle.com/projects/guestvm/shared/guestvm/guestvm/index.html`.

[33] Pike, R., Presotto, D., Thompson, K., and Trickey, H. Plan 9 from Bell Labs. In *Proc. Summer UKUUG Conference* (1990), pp. 1–9.

[34] Porter, D. E., Boyd-Wickizer, S., Howell, J., Olinsky, R., and Hunt, G. C. Rethinking the library OS from the top down. In *Proc. 16th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)* (Newport Beach, California, USA, 2011), ACM, pp. 291–304.

[35] S, G. T. Anandtech: Intel's Haswell NUC: D54250WYK UCFF PC review (`http://bit.ly/1Dxq6hJ`), Jan. 2014.

[36] Scott, D., Sharp, R., Gazagnaire, T., and Madhavapeddy, A. Using functional programming within an industrial product group: perspectives and perceptions. In *Proc. 15th ACM SIGPLAN International Conference on Functional Programming (ICFP)* (Baltimore, Maryland, USA, Sept. 27–29 2010), pp. 87–92.

[37] Simonet, V. The Flow Caml System: Documentation and user's manual. Tech. Rep. RT-0282, INRIA, July 2003.

[38] SOLTESZ, S., PÖTZL, H., FIUCZYNSKI, M. E., BAVIER, A., AND PETERSON, L. Container-based operating system virtualization: A scalable, high-performance alternative to hypervisors. In *Proc. 2nd ACM European Conference on Computer Systems (EuroSys)* (Lisbon, Portugal, 2007), ACM, pp. 275–287.

[39] THIBAULT, S., AND DEEGAN, T. Improving Performance by Embedding HPC Applications in Lightweight Xen Domains. In *2nd Workshop on System-level Virtualization for High Performance Computing (HPCVIRT'08)* (Glasgow, Scotland, Mar. 2008).

[40] VON EICKEN, T., CHANG, C., CZAJKOWSKI, G., HAWBLITZEL, C., HU, D., AND SPOONHOWER, D. J-Kernel: A capability-based operating system for Java. In *Secure Internet Programming, Security Issues for Mobile and Distributed Objects* (1999), J. Vitek and C. D. Jensen, Eds., vol. 1603 of *Lecture Notes in Computer Science*, Springer, pp. 369–393.

[41] VRABLE, M., MA, J., CHEN, J., MOORE, D., VANDEKIEFT, E., SNOEREN, A. C., VOELKER, G. M., AND SAVAGE, S. Scalability, fidelity, and containment in the Potemkin virtual honeyfarm. In *Proc. 20th ACM Symposium on Operating Systems Principles (SOSP)* (Brighton, United Kingdom, 2005), ACM, pp. 148–162.

[42] WARFIELD, A., FRASER, K., HAND, S., AND DEEGAN, T. Facilitating the development of soft devices. In *Proc. USENIX Annual Technical Conference (ATC)* (Apr. 10–15 2005), pp. 379–382.

[43] ZHANG, X., MCINTOSH, S., ROHATGI, P., AND GRIFFIN, J. L. XenSocket: A high-throughput interdomain transport for virtual machines. In *Proc. ACM/IFIP/USENIX International Conference on Middleware (Middleware)* (Newport Beach, California, 2007), Springer-Verlag New York, Inc., pp. 184–203.

Tardigrade: Leveraging Lightweight Virtual Machines to Easily and Efficiently Construct Fault-Tolerant Services

Jacob R. Lorch, Andrew Baumann, Lisa Glendenning*, Dutch T. Meyer†, and Andrew Warfield†
*Microsoft Research, *University of Washington, †University of British Columbia*

Abstract

Many services need to survive machine failures, but designing and deploying fault-tolerant services can be difficult and error-prone. In this work, we present Tardigrade, a system that deploys an existing, unmodified binary as a fault-tolerant service. Tardigrade replicates the service on several machines so that it continues running even when some of them fail. Yet, it keeps the service states synchronized so clients see strongly consistent results. To achieve this efficiently, we use *lightweight virtual machine replication*. A lightweight virtual machine is a process sandboxed so that its external dependencies are completely encapsulated, enabling it to be migrated across machines. To let unmodified binaries run within such a sandbox, the sandbox also contains a library OS providing the expected API. We evaluate Tardigrade's performance and demonstrate its applicability to a variety of services, showing that it can convert these services into fault-tolerant ones transparently and efficiently.

1 Introduction

Tolerating machine failure is a key requirement of many services, but achieving this goal remains frustratingly complex. Many services that do not otherwise require a distributed system must, in some form, consistently replicate the critical aspects of their system state on different hosts. This requirement is particularly burdensome for simple applications that could, if not for the risk associated with a single point of failure, be deployed on a single machine running commodity software.

Tools exist to support developers writing fault-tolerant services, such as replicated state machine libraries [6, 22] and coordination services for consistent metadata storage [5, 18]. However, even when this is possible, supporting the semantics of these tools requires the efforts of expert systems designers, and puts significant demands on the service's design.

A promising alternative is asynchronous virtual machine replication (VMR), as used in the Remus system [11]. This approach transparently protects an arbitrary service by running it in a replicated VM. Externally observable consistency is achieved by buffering network output until a checkpoint of the system state that created the output has been replicated. Output buffering means that client-perceived latency will increase as the time to capture and disseminate a checkpoint increases, motivating techniques to reduce the size of these checkpoints.

To address this, we introduce the concept of asynchronous *lightweight* VM replication (LVMR), which uses lightweight virtual machines (LVMs) in place of VMs [3, 30]. A lightweight VM provides encapsulation with a smaller memory footprint because background operating system services are outside of the container. This substantially reduces the time to create and replicate checkpoints, leading to a reduction in both service latency and replication bandwidth.

An LVM has a higher-level interface between guest and host than a VM, so some techniques used in VMR do not directly translate. We implement LVMR as an extension that interposes on an existing, general binary interface between an LVM guest and host. This requires dealing with non-determinism in the interface, using existing calls to quiesce the system so a consistent snapshot can be captured, and checkpointing through the interface.

To demonstrate the practicality of our design, we implement it as a system we call Tardigrade. We show that, through reasonable optimizations like in-memory checkpointing, identification of hot pages, and delta encoding, the cost of checkpointing can be made low. We find that client-perceived latency impact for a simple application is $\sim$11 ms on average, with a 99.9th quantile latency under 20 ms. Furthermore, this latency does not skyrocket when external processes like OS updates run on the host.

Tardigrade uses primary-backup replication to survive as many faults as there are backups. These faults must be external to the service, e.g., power loss, disconnection, or system crash, since replication cannot mask faults that cause the primary to corrupt replicated state. Instead of relying on synchrony assumptions, we use a variant of Vertical Paxos [20] for automatic failure recovery. This permits the use of an unreliable failure detector to decide when to fail over the active replica, and allows replicas to communicate over a standard network.

A key design goal of Tardigrade is that it permits simpler design of fault-tolerant systems. We demonstrate this by encapsulating and evaluating three existing services that were developed without fault tolerance as a first concern.

In summary, the contributions of this paper are:

- We introduce the idea of asynchronous LVM replication (LVMR) and describe a complete design of a system supporting it.
- We illustrate the practicality of our design by implementing it in the Tardigrade system, applying optimizations to make it performant, and evaluating the resulting performance.
- We demonstrate how LVMR makes it easy to deploy fault-tolerant services. To show this, we replicate the FDS metadata service [27], ZooKeeper, and Apache without changing their binaries.

2 Background and Motivation

In this section, we provide background needed to understand the motivation and design of our system. First, §2.1 evaluates the cost of replicating OS background state in VMR. §2.2 describes the concept of a lightweight VM (LVM), which we use in Tardigrade. Finally, §2.3 overviews the specific LVM system used by Tardigrade, Bascule [3], which we chose because of its extensibility.

2.1 Overheads in Asynchronous Virtual Machine Replication

Remus [11] introduced asynchronous virtual machine replication (VMR). In VMR, the protected guest software is encapsulated in a VM, and snapshots of its state are frequently sent to a backup host across the network. On the backup, the VM image is resident in memory and begins execution immediately upon primary failure.

The amount of time the guest is suspended during the snapshot is minimized using *speculative execution* [26], in which the guest executes while the most recent snapshot of its state is asynchronously replicated. To prevent externally-observable inconsistencies, Remus buffers the output of speculative execution, i.e., network packets, and releases it only when the state that produced it is durably replicated. This mechanism bounds the minimum latency of the system observed by clients by the amount of time required to take and replicate a snapshot.

To understand these overheads, we test two Xen-based hosts connected by a 1 Gb/s network. We use RemusDB, the highest performing version of Remus available, to protect a Windows Server 2012 guest.

First, we measure the cost of the suspend/resume operation Remus uses in isolation—without replication or network buffering—on an idle guest. We find it is 10 ms regardless of checkpoint interval. This is due to the overhead of suspending an unmodified guest VM, which requires the guest and each virtualized device to synchronize its internal state, e.g., flushing processor caches to RAM via an ACPI interrupt. Note that Linux can be paravirtualized to perform this synchronization much more quickly, in <1 ms.

Next, we measure the effect of common OS background tasks on latency. Figure 1 evaluates ping response time for both an idle baseline and when a single background OS task is running. We set a 50-ms checkpoint interval for each case. This is a conservative choice. While a lower interval would provide improved latency, the benefits would be seen across all configurations, and the workload throughput would suffer because the experiment would spend more time in a suspended state.

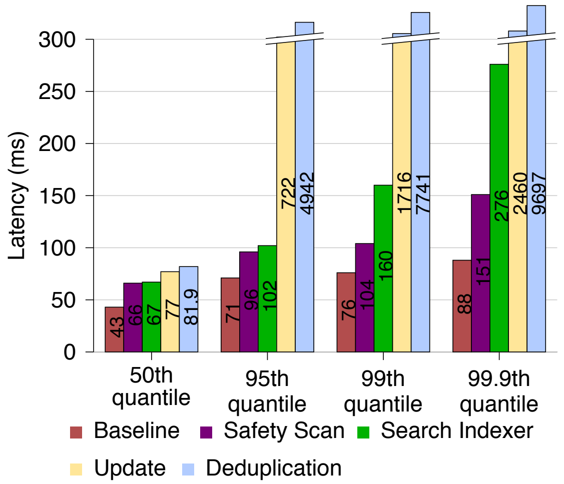


Figure 1: Effect of background tasks on ping latency of Windows Server 2012 under Remus protection

The four background tasks we evaluate are common and important services for Windows file servers:

1. *Safety Scanner* protects the OS against malware.
2. *Search Indexer* manages an index of file contents and properties to optimize lookups.
3. *Windows Update* fetches and applies critical patches to the OS.
4. *Single Instance Storage* deduplication improves storage efficiency.

The most costly background activity we measure is deduplication, which shows maximum incremental checkpoint sizes of 691 MB after compression. This causes delays of more than seven seconds to commit states, during which all communication is buffered. This is the primary contributor to the high ping times observed in all tests. Even Safety Scanner, which dirties memory at the most modest rate, still produces a more than 50% increase in median latency.

These results support the intuition that checkpointing the state of non-critical OS background services in a VM has a significant cost in both replication bandwidth and service latency.

2.2 Lightweight VMs

A traditional virtual machine provides the abstraction of a dedicated machine complete with kernel mode, multiple address spaces, and virtual hardware devices. It is therefore able to run traditional OSes, perhaps lightly modified for paravirtualization, and can host multiple applications. In contrast, a lightweight VM (LVM) is constructed from a single isolated user-mode address space, referred to as a *picoprocess* [13]. An LVM typically runs only a single application along with a library OS (LibOS), which provides the application with the APIs on which it depends. Past work on Drawbridge [30] refactored an existing monolithic OS, Windows, to create a self-contained LibOS running in a picoprocess yet still supporting rich desktop applications.

Despite being able to run unmodified applications, an LVM typically has substantially lower overhead than full VMs. This is because it elides most OS components not needed to implement application-facing interfaces, such as the file system, device drivers, and service processes. For example, the Drawbridge authors reported a disk footprint of 64MB and working set of 16MB for their Windows 7 LibOS [30]. Compared to typical VM footprints measured in gigabytes, this small scale makes lightweight VMs attractive for efficient replication.

2.3 Bascule

Bascule [3] is an architecture for LibOS extensions based on Drawbridge. It defines a narrow binary interface, the *Bascule ABI*, consisting of 40 downcalls and 3 upcalls implementing primitive OS abstractions: virtual memory allocation and protection, exception handling, threads and synchronization mechanisms, and finally an I/O stream abstraction used for files and network sockets. All interaction between a LibOS and the host must traverse the ABI. Bascule extensions, such as our checkpointer (§3.3), are loaded in-process with the LibOS and application, and interpose on the ABI. Since the Bascule ABI is designed to be independent of host OS and guest LibOS, and to enable arbitrary nesting of implementations, extensions support a variety of platforms, and may be composed at runtime.

3 Design

This section presents the design of Tardigrade. We start with an architectural overview, then discuss each of the pieces of that architecture in turn. Our design assumes a fail-stop model: server machines will fail only by stopping, not by acting arbitrarily.

3.1 Overview

Figure 2 illustrates the architecture of the Tardigrade system. On each machine, we run an *instance* that will at various times act as a primary service replica, a backup service replica, or a spare. The *orchestrator* coordinates these instances to ensure they act in concert as a consistent, fault-tolerant service, using a variant of Vertical Paxos [20]. The orchestrator uses an unreliable failure detector to get hints about which instances have failed.

Each instance makes use of two subcomponents to do its job: a Bascule component consisting of a host and guest, and a *network filter*. The Bascule component runs the service in a lightweight VM. The network filter only releases guest output when the checkpoint of a state following its generation has been durably replicated.

The Bascule guest contains, like typical Bascule guests, an unmodified application running atop a library OS mimicking the OS the application expects. We leave these components unchanged, and add a *checkpointer* below the library OS that lets the Bascule guest checkpoint its state or restore its state from a checkpoint.

3.2 Orchestration

The orchestrator manages the instances using the Vertical Paxos protocol [20] and is divided into two components: the unreliable failure detector and the view manager. Note that our terminology differs slightly from that of Vertical Paxos: we call the master an *orchestrator* and call ballots *views*.

A *checkpoint* is a snapshot of the LVM's state. A *full* checkpoint is a self-contained checkpoint, while an *incremental* checkpoint reflects only changes that have been made to the LVM during an inter-checkpoint interval, also known as an *epoch*. An incremental checkpoint thus describes how to go from a *pre-state* to a *post-state*.

A *view* is an assignment of roles to instances; instances can take on three roles. When *primary* it runs the service and responds to client requests. When *backup* it records checkpoints of the primary's state so that it can become primary if needed. When *spare* it simply waits until it is needed as a primary or backup.

The primary of the first view starts a fresh LVM and disseminates a full checkpoint, then transitions to periodically taking incremental checkpoints. Checkpointing involves the following steps: *quiescing* the guest, capturing the checkpoint, resuming guest execution, and sending the checkpoint to backups.

A received checkpoint is *applicable* at a backup if the backup can restore it. A full checkpoint is always applicable, and an incremental checkpoint is applicable if the backup can recreate the incremental checkpoint's pre-state. For instance, if a backup has a full checkpoint and the three incremental ones following it, then all four of these are applicable.

Once a checkpoint is applicable at all backups, it is *stable*. That is, as long as one of the backups or the primary remains alive, the system can proceed.

When the primary learns that a checkpoint is stable, it *decides* the checkpoint. That is, it considers the checkpoint to describe the next official state in the sequence of service states. Thus the service's logical lifetime is divided into epochs punctuated by decided checkpoints; we call an epoch decided when its ending checkpoint is decided. Once an epoch has been decided, it is safe to send any network packets the primary generated during that epoch, because it will never be necessary to roll back to an earlier state.

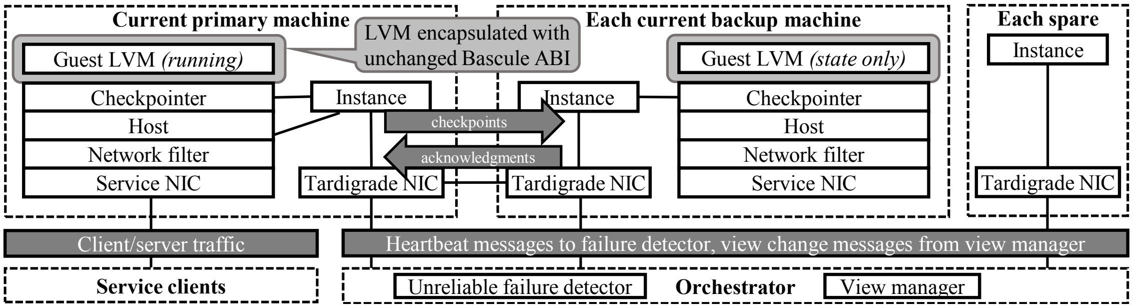


Figure 2: Overview of Tardigrade architecture for replicating lightweight virtual machines (LVMs)

When a backup learns that a checkpoint is decided, it can *apply* the checkpoint to its state. For a full checkpoint, the backup starts a new LVM and initializes its state to match. For an incremental checkpoint, the backup performs the operations necessary to transform the current pre-state into the post-state. We implement a queue of applicable checkpoints at a backup that is processed asynchronously with sending acknowledgements to the primary; the queue size tuning knob balances latency of checkpoint acknowledgements in the steady state with recovery time in the relatively rare failover event.

Each instance periodically sends a heartbeat to the orchestrator. After detecting a primary or backup failure, the orchestrator proposes a new view, which includes the identity of the new primary and backups. The new primary is, if available, the primary from the previous view; otherwise, it is a backup from the previous view.

If the new primary was previously a backup, it stops accepting checkpoints from the previous view and finishes applying checkpoints it has acknowledged. The new primary then takes a full checkpoint, disseminates it to the new backups, then asks the orchestrator to *activate* the view. The orchestrator activates the view by deciding that it follows the previous active view; then, the new primary considers the initial checkpoint stable and proceeds with further checkpoints.

When a backup is elevated to primary, the initial state the backup uses is guaranteed to match a state of the previous view's primary at or beyond the last state the primary knew to be stable. However, in the case it is beyond the last state the primary knew to be stable, the primary will not have released the network traffic generated between its last stable checkpoint and the backup's initial state. Note that this same technique was used in Remus [11]; the insight is that losing the network output of state that was successfully replicated mimics the case of the actual network dropping packets, and services are typically written to be robust to unreliable networks.

When a spare gets a message with the initial full checkpoint of a new view, it saves and acknowledges the checkpoint. When the primary later tells the spare that the view is activated, the spare becomes a backup.

Note that an orchestrator may propose a new view but never activate it. If machines fail during the view change, the orchestrator may propose a different new view to succeed the current view. It will eventually activate only one, and the initial checkpoints of the aborted views can be discarded.

3.3 Checkpointer

Our checkpointer is designed as a Bascule extension and is responsible for both capturing and applying checkpoints. In Bascule, a guest LibOS uses a PAL to translate the guest's ABI calls into underlying system calls. The checkpointer extends the system by interposing between the two. In other words, it provides the Bascule ABI to the guest, and satisfies the requests the guest makes by passing them on to the PAL. From this position, it can track all system objects (e.g., files, threads, synchronizers) that the guest uses, and it can virtualize system objects so that they are portable across machines.

A naive checkpoint would be a list of all ABI calls made by the guest and a snapshot of its CPU state and memory contents. However, this is impractical for two reasons. First, it would produce extremely large checkpoints. Second, ABI calls are not deterministic, so just replaying them will not necessarily bring about the same state on the new machine. Instead, the checkpointer inspects the current state and produces a list of actions that can reproduce that state.

3.3.1 Memory tracking

The checkpointer tracks the following information for each memory region allocated by the LVM: location, protection, and which pages may have been modified during the current epoch. This metadata is stored in an AVL tree where each node represents a memory region.

Identifying a subset of memory that has not been modified in the preceding epoch is essential for generating efficient incremental checkpoints. Ideally the checkpointer would use hardware such as page-table dirty bits for this, but it is not accessible through the Bascule ABI. Instead, the checkpointer uses a standard technique for tracking memory modifications. First, before each epoch the checkpointer write-protects all writable pages using the `VirtualMemoryProtect` ABI call. During the epoch, when the guest writes a protected page, the checkpointer intercepts the triggered access violation exception. The exception handler restores the original page protection, sets the corresponding dirty bit in the metadata tree, and suppresses the exception from the guest.

Many optimizations of this general design are possible; §4.1 discusses the optimizations we implemented.

One complication is that although we can suppress access violation exceptions incurred by the guest, we cannot suppress exceptions incurred by the host. This can happen when the guest passes the host a pointer to memory the checkpointer has write-protected. So, before passing any guest pointers to the host, the checkpointer ensures that they are not write-protected. For pointers to small objects, like an integer, the checkpointer simply substitutes pointers to its own stack, and then copies the values into the guest-supplied pointers when the call returns. For pointers to large objects, like a buffer, the checkpointer touches all the pages in advance so that any exceptions are incurred by it rather than the host.

3.3.2 File-change tracking

For each mutable private file system (FS) of the LVM, the checkpointer tracks which parts have potentially changed during the last epoch. For each file, the checkpointer tracks possible changes to its existence, its metadata, and its blocks. Note that it does not track the actual contents of those changes, as they can be read directly from the FS during checkpointing.

Operations that can potentially change a file include open, delete, rename, map, and write. However, all write ABI calls are asynchronous, so the checkpointer tracks changes due to a write only when the call has completed. Before a write completes, a checkpoint will capture the ongoing write to replay on restore, so there is no need to capture the actual change to the file. For similar reasons, the checkpointer does not track changes due to mapped files until the region is unmapped.

3.3.3 Quiescence

To take a consistent snapshot of an LVM's state, each of its threads must first *quiesce*, i.e., pause so that it stops mutating state. Additionally, because the Bascule ABI does not provide a way for one thread to capture another's state, each quiescing thread must also capture its own state with standard x86 instructions and store it in a location accessible to the checkpointer.

We use different methods to quiesce a thread, depending on which of three states it is in: the middle of a blocking ABI call to the host, in the middle of a non-blocking but nevertheless uninterruptible operation, or neither. A thread in the latter state is quiesced by raising an interrupt in the thread; the checkpointer's interrupt handler will initiate quiescence. Handling the first two states is more involved.

Before a thread enters a non-blocking but uninterruptible state, such as a non-blocking ABI call or a code section that mutates checkpointer-tracked state, the thread acquires a *checkpoint guard*. This is essentially a per-thread lock that is re-entrant since a thread holding a guard may take an exception that itself requires a guard. We implement the checkpoint guard as a simple atomic counter. If a quiescence interrupt occurs while a thread holds the guard, then the interrupt is ignored and the thread sets a flag to quiesce when the guard is released. This will happen shortly, since by assumption the thread is in the middle of only non-blocking operations.

The final case to consider is when the thread has entered a blocking ABI call. Fortunately, there are only two indefinitely blocking ABI calls: `ObjectsWaitAny`, which waits for one of an array of handles to be signaled, and `StreamOpen`, which can block when asked to open an outgoing TCP connection.

When the guest calls `ObjectsWaitAny`, the checkpointer adds an additional handle to the list of handles to be waited on; this extra handle is to the *quiescence-requested* event. When the checkpointer initiates quiescence, it sets this event, thereby waking any such blocked threads. When a thread returns from `ObjectsWaitAny`, it quiesces if the quiescence-requested event is set. Since the wait call was prematurely terminated, the thread repeats the call upon resuming; if the wait had a relative timeout, then the thread reduces it by the amount of time already spent waiting and/or checkpointing.

When the guest calls `StreamOpen` with parameters for opening an outgoing TCP connection, the thread first captures its state and marks itself as quiesced. The thread then proceeds with the blocking call since it will not hold up any checkpoints that occur during the call. Upon return, the thread waits until any concurrent checkpointing completes, then rescinds its claim to be quiesced and proceeds with execution. Note that this approach would work for arbitrary calls, not just `StreamOpen`, that do not mutate guest state. However, it requires an expensive thread capture on each call, so we do not use it for `ObjectsWaitAny` where a more lightweight solution exists.

3.3.4 Dealing with non-determinism of Bascule ABI

The Bascule ABI has several sources of non-determinism. The checkpointer must hide them so that restoring a checkpoint results in a replica of the checkpointed state.

The simplest and most widespread source of non-determinism is handle identifiers. Because the host can assign arbitrary identifiers to handles, there is no guarantee it will assign the same ones during restoration of a checkpoint as were used at the time of checkpoint. So, the checkpointer virtualizes handles by maintaining a mapping between guest virtual handles and host handles and by translating handles in ABI calls.

A subtle source of non-determinism is address space layout randomization (ASLR) [29, 35]. Host ASLR can rearrange the contents of a read-only binary file, so even if a primary and backup have duplicate file contents they may diverge. To handle this, the checkpointer interposes on the guest ABI call that maps binary files. Before returning to the guest, the checkpointer performs all necessary relocation to reflect the address where the file actually got mapped. In other words, the checkpointer ensures the guest's binary-mapping ABI is deterministic even though the host-provided ABI is not.

One source of non-determinism requires a small change to the ABI. In the original Bascule ABI, when the guest creates an HTTP request queue, the host assigns it a non-deterministic ID that is then used by the guest in subsequent calls to open HTTP requests. We address this by changing Bascule's ABI so that the guest assigns the ID instead. This is the only case where we modify the ABI. Modifying existing host and LibOS implementations to support Bascule's new ABI should be relatively straightforward: our modifications to the Windows host and LibOS constitute only $\sim$250 lines.

3.4 Networking

The IP address clients use to connect to the replicated service is the *service address*. Each instance devotes a NIC to the service that is separate from the NIC it uses to communicate with the orchestrator and other Tardigrade instances. We call this the *service NIC*. Only the primary sends traffic on the service NIC, including ARP packets, so the network and clients see only one machine using the shared address at a time.

To interpose on the service NIC, we implement a network filter that suppresses non-primary output and buffers primary output during epochs. The buffered output of the primary is released only when the following checkpoint is stable.

Buffering interacts with the current ABI in surprising ways, as discussed in §4.3. One consequence is that, until the ABI changes from a socket-based to a packet-based interface, TCP connections are broken on a failover event.

4 Implementation

Our implementation includes the following components, with line counts measured by SLOCCount [34].

- Bascule checkpointer extension: 17,056 lines of C, plus 1,226 lines of Python to produce automatically-generated hook functions (not separately counted).
- Network filter, implemented as a Windows kernel-mode driver: 1,329 lines of C.
- Orchestrator and instance: 683 and 2,718 lines of C#, respectively, plus 1,346 lines of common code used by both for inter-communication.
- Plugin to let instance and checkpointer extension communicate, using Bascule host's support for extending the stream namespace: 2,773 lines of C++.

A limitation of our current implementation is that the orchestrator runs on a single machine, so it is a single point of failure for the system. To improve fault-tolerance, our plan is to divide the orchestrator into two components: the unreliable failure detector and the view manager. The failure detector does not require consistent state, so it can be made fault-tolerant using simple stateless mechanisms; however, the view manager will be redesigned as a state machine and run with a replicated state machine library [6, 22].

The remainder of this section overviews some lessons learned during the implementation of Tardigrade.

4.1 Memory checkpointing optimizations

This subsection describes the optimizations we use to improve the performance of memory checkpointing.

The first optimization reduces checkpoint size by calculating updates to memory at a finer granularity than a page using a twin-diff-delta technique [2]. In this technique, the exception handler that executes when a write-protected page is first written in an epoch stores a copy of the pre-write contents of the page. Then, the checkpoint at the end of the epoch uses delta encoding [17] to capture the difference in the page content more precisely.

The next optimization selectively disables write-protection for hot pages. Our heuristic for deciding that a page is hot is exceeding a threshold for the number of consecutive epochs that the page has been written to, defaulting to three. When write-protection is disabled for a hot page, the checkpointer simply assumes that it is always dirty. However, a side-effect of this mechanism is that the checkpointer cannot detect when the hot page is no longer being written by the guest, so every epoch we flip each hot page to cold with a fixed probability. We found performance to be fairly insensitive to this value of in a broad range; we default to the value $1/16$ which lies within that range. An alternative would be to use twin-diff-delta to detect when a hot page has not been written in a given epoch; we plan to investigate this in future work.

Another important optimization uses parallelism to reduce the time to snapshot memory changes. Our checkpointer maintains several threads, roughly one per core, and disseminates independent memory-snapshotting tasks to them via a shared task queue. We could have parallelized other checkpointing operations besides memory snapshotting, but found it generally not to pay off: other operations are so quick or rare that queuing and scheduling time overwhelms the benefits of parallelization. So, the only other snapshotting operation we parallelize is capturing thread states.

Normally, an epoch ends when (1) the previous epoch's checkpoint has been disseminated to the backups and acknowledged, and (2) the current epoch's duration is greater than some minimum, typically set to zero. Our final optimization, *checkpoint capping*, can end an epoch earlier based on the rate of memory dirtying in the guest. Checkpoint capping mitigates the effect of rapid memory dirtying on increased time to take a checkpoint, which is especially problematic for guests running in platforms with garbage collection. The goal of checkpoint capping is to automatically end the epoch before the resulting checkpoint can get too large. However, during the epoch it is infeasible to efficiently and precisely predict the potential checkpoint size while accounting for additional optimizations like delta encoding. So, the heuristic we use is to prematurely end the epoch once the number of dirtied pages reaches a configurable threshold.

4.2 In-memory checkpoints

We found that writing checkpoints to files on a Windows host dominated the cost of checkpoint capture and dissemination, even when the files are not stored on disk. So, instead of using files, each instance shares a memory region with the checkpointer extension. The checkpointer captures checkpoints directly to this shared memory, and the instance copies checkpoints received over the network directly into it as well. The section defaults to 1 GB; if this is too small for a particular checkpoint it uses a file instead.

4.3 Breaking connections

The Bascule ABI supports networking through a socket interface. To open a TCP or UDP socket, the guest opens a specially-named stream. To send or receive on that socket, the guest writes or reads the stream handle.

This socket-based interface presents a challenge: since TCP session state is in the host rather than in the guest, it cannot be seen or modified by the checkpointer. Therefore, when restoring an LVM on a new machine, the TCP state will be different and the guest will not be able to communicate over existing connections. To address this problem, the checkpointer breaks all TCP connections before restoring the guest. This is implemented by restoring TCP and HTTP streams as a handle to a special event that is always signalled. When the restored guest starts running and calls an operation on such a handle, the operation will return immediately with `STATUS_CONNECTION_RESET`.

Our hypothesis is that services are written to recover from such transient disconnections, and we find that this hypothesis holds for the services evaluated in §5. A cleaner solution would be to modify the ABI to support checkpointing guest networking state so that connections can be transparently migrated across hosts.

This has taught us a lesson about the design of Bascule. The socket ABI was chosen for expedience, since it obviated the need to build a network stack in the LibOS. The ABI designers were aware that this choice was problematic for compatibility and portability; our work on Tardigrade demonstrates that it also interferes with migration. We believe that the path forward for the Bascule networking interface is to use packets rather than sockets as the interface between guest and host, and we are working with the Bascule development team to realize this goal. An additional benefit of a packet-based interface is to enable packet buffering within the checkpointer extension itself instead of requiring an external network filter.

4.4 Network buffering

Network buffering effectively increases the round-trip time of connections to the server, increasing the bandwidth-delay product of each connection. For TCP connections, this necessitates both a large window size and a large buffer for sent but unacknowledged packets. A Windows host detects this high delay and adjusts the TCP window size in response, but it does not automatically increase the send buffer size.

To fix this, we update the `DefaultSendWindow` registry setting so the send buffer size exceeds the expected bandwidth-delay product. Note that this setting should also be managed on any client that sends significant traffic, because network buffering on the server delays acknowledgment of client packets, causing the client to buffer sent packets for up to two epochs.

5 Evaluation

5.1 Methodology

The machines we use in our experiments are Dell PowerEdge R710 rack servers. Each is configured with two quad-core 2.26 GHz Xeon E5520s, 24 GB RAM, two Broadcom BCM5709C NetXtreme II Gigabit Ethernet NICs, and a Seagate Constellation ST9500530NS 500 GB SATA disk. All the NICs are connected to a single 48-port switch.

Except when otherwise specified, we use four machines: the primary and orchestrator. the backup, the spare, and the client. On each machine, Tardigrade uses one NIC and the replicated service uses the other. The client machine runs Windows Server 2008 R2 Enterprise, and the other machines run Windows Server 2012 R2 Datacenter. To minimize latency, we configure the system to checkpoint as frequently as possible, i.e., to initiate a checkpoint as soon as the previous one is stable.

Our evaluation covers a range of microbenchmarks and real-world services. The microbenchmarking experiments use a simple ping server that listens on a UDP

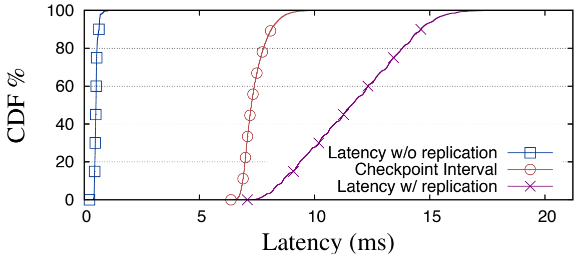


Figure 3: CDF of latency seen by ping client, alongside CDF of checkpoint interval

port and responds with pongs. This ping server can be configured to dirty memory at a given rate by looping through a 100 MiB region one byte at a time, incrementing each byte modulo 256. To ensure the ping service achieves this dirtying rate, the microbenchmarks disable the checkpoint-capping optimization described in §4.1. Also, the memory-dirtying algorithm accounts for real time: e.g., when it is unscheduled, it makes up for the lost time by dirtying memory until caught up. The client of the ping service sends 100,000 requests, one every 2 ms.

5.2 Latency impact

Our first experiment evaluates the base latency overhead of Tardigrade by running the ping server without memory dirtying and measuring the ping response times seen by the client. Figure 3 shows the CDF of this latency.

Running the service in Tardigrade increases the average latency from 0.5 ms to 11.6 ms, but the 99.9% quantile latency is not substantially higher, only 17.5 ms. The proximate cause of this latency is the interval between consecutive checkpoints, whose CDF is also shown in Figure 3. As expected, the average service latency is the baseline service latency plus 1.5 times the average checkpoint interval. After all, if the server sends a packet at time t, Tardigrade will release that packet when the incremental checkpoint covering t is stable, i.e., at the end of the subsequent interval.

We also measured CPU utilization during this experiment to evaluate CPU overhead. We found that the baseline utilization of the unprotected service was 7.3%, that running the service in Bascule slightly increases utilization to 7.9%, and that running it in Tardigrade modestly increases utilization to 13.5%. Most of the observed utilization increase in Tardigrade is from the orchestrator and primary instance processes.

5.3 Effect of dirtying rate on latency

Next, we evaluate the effect of memory-dirtying rate on latency; we expect that higher memory-dirtying rates will increase checkpoint size, thereby increasing the time required to replicate each checkpoint over the network. Figure 4 shows the latency observed by the client as the memory-dirtying rate increases from 0% to 50% of the network speed, i.e., from 0 to 512 Mb/s.

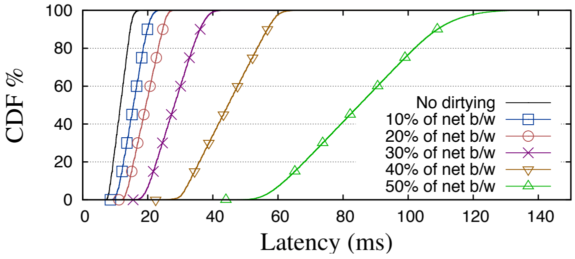


Figure 4: Effect of memory dirtying rate on CDF of latency seen by ping client

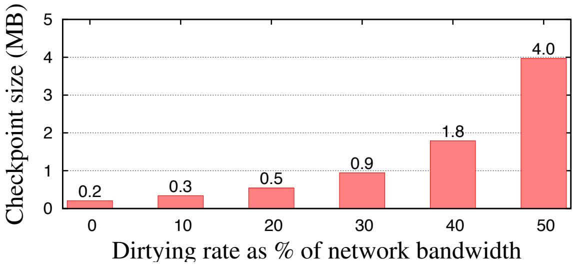


Figure 5: Effect of memory dirtying rate on average checkpoint size of ping server.

When the dirtying rate is 10% of the network bandwidth, i.e., 100 Mb/s, the client latency is reasonable, even at the 99.9th quantile. However, as the rate of memory dirtying rises, the latency seen by clients rises nonlinearly. Indeed, Figure 4 only goes to 50% because a dirtying rate of 60% gives average latency over half a second.

This latency is not due to CPU time. The primary machine's CPU utilization generally decreased as the dirtying rate increased, presumably since more time was spent waiting for the network and the backup.

Figure 5 shows the cause: as expected, average checkpoint size increases as dirtying rate increases. We see that the non-linearity of the increase in client latency tracks the non-linearity of the increase in average checkpoint size. This non-linear effect occurs because larger checkpoints take longer to disseminate, leading to to longer periods the service running asynchronously with checkpoint dissemination, which leads to even larger checkpoints. Note that this feedback loop stabilizes to an equilibrium; we do not see it increase with time and cause the distribution to diverge. We expect equilibrium as long as the memory-dirtying rate does not exceed the rate of checkpoint capture and dissemination.

These results suggest that asynchronous replication is a poor fit for workloads with sustained memory-dirtying rates that are a significant fraction of the network bandwidth. Fortunately, as shown later in this section, there are useful services with tractable memory-dirtying rates.

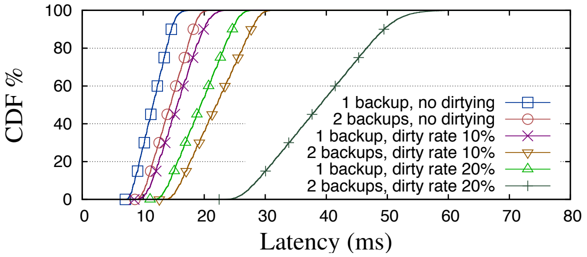


Figure 6: Effect on client latency of adding a backup. Memory-dirtying rate is represented as a percentage of network bandwidth.

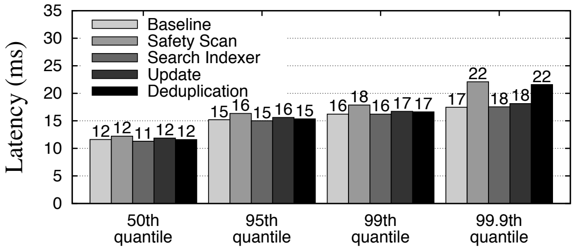


Figure 7: Effect of external processes on latency seen by ping client. Format mirrors that of Figure 1.

5.4 Effect of additional backup

Tardigrade can use multiple backups to tolerate overlapping failures of more than one machine; however, replicating to multiple backups increases the time needed for a checkpoint to be stable. To evaluate this effect, we measure the effect on latency of running the ping service with two backups instead of one. Note that our current implementation does not use IP multicast; if it did, this effect would be significantly reduced.

Figure 6 shows ping latency as a function of both dirtying rate and number of backups. Without dirtying, adding a backup increases latency by only a few milliseconds, which is still quite manageable. With a dirtying rate equivalent to 10% of the network bandwidth, the latency increase is higher than that incurred when adding the same amount of memory dirtying using a single backup. This non-linearity occurs for the same reason as in §5.3; as in those experiments, this acceleration reaches a stable equilibrium: we do not see the checkpoint interval increase with time.

5.5 Impact of external processes

Next, we evaluate the latency impact from resource-intensive processes running on the host, external to the LVM. This experiment uses the ping server without memory dirtying. Recall that we evaluated Remus using this experimental setup in §2, with results shown in Figure 1.

As expected, Figure 7 shows that the impact of external processes on the latency of a service running in Tardigrade is dramatically reduced compared with running in Remus. Indeed, the impact is nearly undetectable for the 99th quantile and below, and hardly noticeable at the 99.9th quantile. We find that external processes cause occasional higher checkpoint periods, likely due to scheduling contention. However, the checkpoint sizes remain unaffected, resulting in 99.9th-quantile latencies in Tardigrade under 25 ms despite external processes that caused 99.9th-quantile latencies of multiple seconds in Remus.

5.6 Failover time

To evaluate the time to recover the service when an instance fails, we use a variant of the client that measures failover times as a long period with no response, i.e., a period with zero service bandwidth. We run the experiment 100 times in each of two scenarios: primary failure and backup failure.

The median recovery time is 500 ms in the backup-failure scenario and 700 ms in the primary-failure scenario. The difference between these two scenarios reflects the time for the new primary to start running; because we keep each backup's in-memory state and objects up to date, this startup cost is only 200 ms. The remaining time is largely due to the 100 ms failure-detection timeout and the time to take a full checkpoint and disseminate it to the new backup. The median size of this full-checkpoint transfer is 26.9 MB, which the new backup takes 225 ms to download.

Note that many services will have larger memory footprints and thus commensurately longer failover times. For instance, the remainder of this section evaluates three real services in Tardigrade. Interruption with a failure at a random point induced full checkpoints of 36 MB for the metadata service, 170 MB for the coordination service, and 636 MB for the web service. Sending the latter over a 1 Gb/s link would take at least 5 s.

An operation that is not on the critical path for recovery is the new backup initializing its LVM. This is because we let the backup acknowledge checkpoints, including full ones, after queueing them for later application. When we prevent this by substantially decreasing the queue size, median response time for the backup-failure scenario becomes 8.8 s. This high figure demonstrates that checkpoint queueing and keeping the backup up-to-date significantly reduce failover delay.

5.7 FDS metadata service

In this and the following two subsections, we evaluate Tardigrade's performance on real services. First, we evaluate a custom configuration metadata service written by colleagues for the FDS research project [27]. In normal operation, this service experiences low traffic because it simply sends and receives periodic heartbeats and informs clients and disk servers when failures occur.

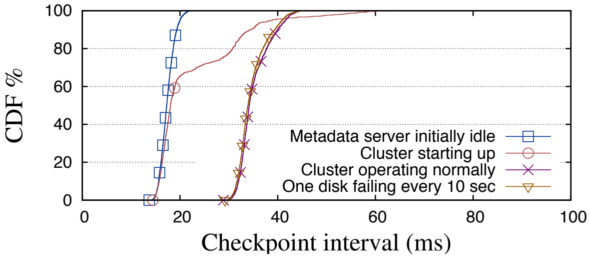


Figure 8: Distribution of checkpoint periods when running the FDS metadata service, during various phases

This experiment uses the FDS cluster's 10 Gb/s network so that the FDS cluster can run as normal but use our fault-tolerant metadata service. However, Tardigrade still uses a 1 Gb/s port for checkpoint dissemination so that the results are comparable to other results presented in this paper.

This experiment proceeds in several phases. First, the metadata service starts up and then idles for one minute. Then, the FDS cluster with 70 disk servers starts up, each of which must communicate with the metadata service to receive a role assignment. Then, ten clients begin executing FDS's stock write-intensive load-testing tool, and the cluster runs normally for two minutes. Finally, we kill one disk server process every ten seconds, provoking the metadata service to react to the departures.

Figure 8 shows the resulting distribution of checkpoint interval. Initially, the checkpoint interval averages 17.4 ms, reflecting an average checkpoint size of 0.9 MB. As the cluster comes online and requests assignments, the checkpoint interval increases but never goes above 64 ms. When the cluster is up and handling requests from disk servers and clients, checkpoint interval maintains a modest average of 35.2 ms, reflecting checkpoint sizes averaging 1.8 MB. This low activity is not surprising, since FDS was designed to reduce load on its metadata service by caching the metadata at participating parties. It is thus an ideal candidate for Tardigrade.

5.8 ZooKeeper coordination service

Our next real service is ZKLite, a custom in-memory implementation of the ZooKeeper server API, written in Java by one of the co-authors for a separate research project. We initialize the server state for the benchmark by creating a balanced binary znode tree of depth 10. The benchmark then executes 100,000 operations, where each operation either reads or writes the data of a random znode. Writes are done with probability $1/3$, and write a uniformly random amount of data between 0 and 10 KB. Operations are launched in parallel, with at most 100 outstanding at once. We report results for the last 90% of operations to reflect steady-state performance.

Since this service is written in a garbage-collected language, it experiences occasional periods of fast memory dirtying. So, we enable checkpoint capping as discussed in §4.1. A key parameter here is the number of pages dirtied in an epoch before triggering quiescence. If this parameter is low, the service spends a lot of time quiesced instead of processing client requests. On the other hand, a high parameter value results in large checkpoints and thus increased buffering time for outbound packets. Both manifest as high client latency. Figure 9 shows the effect on latency of various parameter settings; infinity means that checkpoint capping is turned off. We see that performance is improved substantially by the use of checkpoint capping, is best when quiescence occurs after about 1,000 pages dirtied per epoch, and is fairly insensitive to this parameter value over the range 750–1,500.

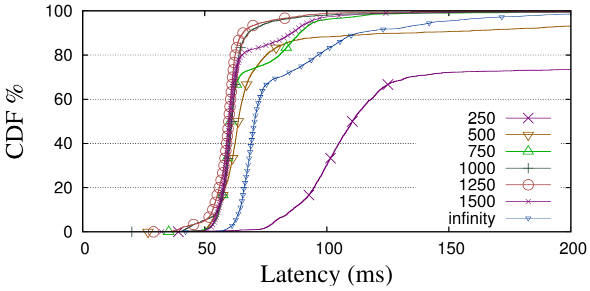


Figure 9: Effect on client latency distribution for ZKLite of number of pages dirtied per epoch before quiescence

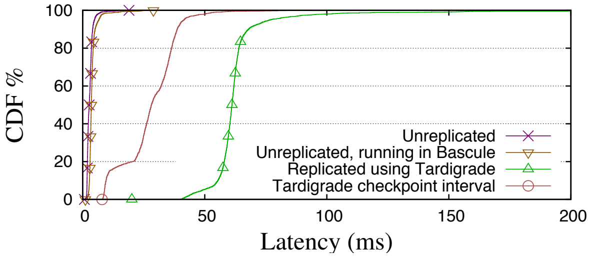


Figure 10: Client latency distribution for ZKLite

Note that other services and workloads may have different ideal values for this parameter. For instance, if a service has low load, it can accommodate being frequently unscheduled, and thus may perform better with a low cap. If a service has high baseline latency, then the effect of network buffering will be relatively inconsequential, and a higher cap may be best.

Having established what parameter to use for checkpoint capping, we compare performance under Tardigrade to baseline performance. Figure 10 shows the results of benchmarks run under three setups: unreplicated; unreplicated, but running in Bascule; and in Tardigrade. The Bascule-only line shows that the overhead of running in Bascule contributes little to the higher latency seen, so as expected it is asynchronous replication that contributes most to latency.

As discussed in §5.2, outbound buffering causes de-

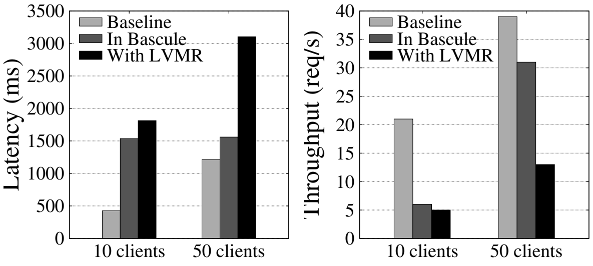


Figure 11: Performance of MediaWiki in Apache with different numbers of client threads

lay of 1.5 times checkpoint interval duration. That duration, also shown in Figure 10, accounts for most of the increased latency; the rest is due to overhead of replication such as handling access-violation exceptions due to memory tracking. The effect of checkpoint capping manifests at the high end of the latency distribution; during periods of high memory dirtying, the service is frequently quiesced, delaying client requests.

The lessons we draw are as follows. Tardigrade can replicate ZKLite under modest load, but at a noticeable latency cost, on the order of 60 ms. The service's use of garbage collection leads to periods of high memory dirtying, which temporarily cause even higher latency. Checkpoint capping can mitigate this problem but not eliminate it, by reducing the time spent buffering packets at the cost of delaying execution of the service.

5.9 Web service

Our web service is the popular MediaWiki running on Apache. Its use of dynamic PHP-generated pages instead of static content is typical for a modern website, and stresses Tardigrade by causing mutation of the service's in-memory state. We use Apache version 2.4.7, PHP v5.5.11, and MediaWiki v1.22.5 backed by a SQLite database. We enable the Alternative PHP Cache for intermediate code and MediaWiki page data.

We operate the server in three modes: normal, within Bascule, and within Tardigrade. We benchmark the service using multiple worker threads on the client, each of which repeatedly fetches the 14-KiB main page over a persistent HTTP connection, waiting for the completion of each fetch before initiating the next. We measure the system once it has reached steady state.

Figure 11 shows results under two load conditions: either 10 or 50 client threads. We see that some of the overhead of Tardigrade comes from running in an LVM and some is due to replication. In particular, the Bascule LVM adds significant latency to this workload because each request issues many small file I/Os, primarily `stat` calls, each of which requires an RPC to a separate security monitor process. This overhead is effectively amortized through batching and pipelining at 50 client threads, giving a significant increase in throughput with little added latency. In contrast, the latency overhead due to replication does increase with load: as load increases, so does the memory-dirtying rate and thus the checkpoint size and interval. With 10 client threads, the checkpoint interval average is 54.4 ms, but with 50 client threads it balloons to 475 ms. We conclude that web services may need modest load to be amenable to LVMR.

5.10 Complexity of services

The real services we use require no modifications to run under Tardigrade, supporting our hypothesis that Tardigrade can make unmodified binaries into fault-tolerant services. In the cases of the FDS metadata service and ZKLite, this also supports our hypothesis that Tardigrade can reduce code complexity and developer effort. According to the CodePro plugin for Eclipse, ZKLite is 24,082 lines of code, less than the 30,889 lines for the Apache ZooKeeper server. The smaller count reflects the fact that ZKLite does not have any code for dealing with failures. Further, FDS's metadata service was written two years ago on the assumption that someday it could be rearchitected for fault tolerance, but in that time no one at Microsoft has found the time to do so. Running it within Tardigrade makes the service fault-tolerant with no developer effort.

6 Discussion and Future Work

§6.1 distills the results of our evaluation into a categorization of services that may be good candidates for LVMR. Then, §6.2 discusses directions for future work.

6.1 Candidate service characteristics

Providing fault tolerance at the virtualization layer saves development effort at the expense of runtime performance. Based on our evaluation, we offer some guidance on the classes of applications for which the overhead of Tardigrade may or may not be reasonable.

An important characteristic to consider is the rate and magnitude of memory dirtying. If the memory-dirtying rate is significant relative to the network bandwidth, then LVMR will spend too much time taking checkpoints and transmitting them to backups. Our evaluation suggests that memory-dirtying rates above ∼40% of network bandwidth will cause significant delays. Also, as shown in §5.8, occasional bursts of memory dirtying, e.g., due to garbage collection, manifest as occasional periods of high latency even with checkpoint capping.

There are other reasons a service may not be a good candidate for LVMR. If a service must remain available despite software bugs within the service itself, then LVMR is not applicable. Also, if the service can tolerate very high latencies, then LVMR's main benefit relative to VMR is moot.

One class of promising candidate services for LVMR is metadata and coordination services. These critical services are usually required to be both highly available and strongly consistent because entire distributed systems depend on them. Also, because these services tend to be centralized and therefore a potential bottleneck, system designers often use techniques such as client caching and coarse-grained synchronization to minimize service load. Another favorable characteristic is that the rate of state mutation in such services tends to be low; e.g., the ratio of read to write operations in a typical ZooKeeper workload varies between 10:1 and 100:1 [18].

Another class of good candidate services is niche web applications with a small number of users, e.g., web sites internal to an organization such as requisitioning systems and charity event managers.

An example of a service that is not a good candidate for LVMR is a DBMS. We ran SQL Server inside Tardigrade but found its performance to be poor due to large checkpoints. For such workloads, customization may be necessary, as done by RemusDB [24].

6.2 Future work

Our experience building Tardigrade has highlighted areas for improvement in the underlying LVM technology, Bascule. For instance, we discussed the difficulties caused by having a non-deterministic ABI in §3.3.4 and how Bascule would be improved by offering a packet-based network interface in §4.3. We hope that these lessons can inform the design of lightweight process container technologies to support migration.

There are a number of potential future directions for Tardigrade. First, we believe we can improve performance of guests by making slow operations appear to complete before they actually have, as done by Speculator [26]. Tardigrade already supports buffering output until a speculative operation has completed, and could be extended to support rollback in the case of operation failure. A second direction is exploring how to tune the library OS to improve Tardigrade performance: essentially, this would be to LVMs what paravirtualization [33] is to VMs. Third is grouping multiple LVMs into checkpoint domains to coordinate checkpointing among them, thereby essentially performing distributed snapshots [7] for LVMs. This may improve performance as we would not have to buffer network traffic between LVMs in the same checkpoint domain.

7 Related Work

This section first surveys related work that transparently provides fault tolerance using encapsulation, then briefly discusses alternate approaches to designing and implementing fault-tolerant services.

Encapsulation-based fault-tolerance

The key insights of Bressoud and Schneider [4] were that a VM is a well-defined state machine, and that implementing state machine replication [32] at the VM level is attractive in terms of engineering and time-to-market costs compared to implementations at the hardware, operating system, or application levels. Their system enforced deterministic execution of a primary and backup VM in lock-step through capture and replay of input events by the hypervisor.

VMware's server virtualization platform vSphere [31] provides high availability for a VM using primary-backup replication. Supporting multiprocessors can incur a high performance cost in a deterministic record-and-replay approach; instead, the most recent release of vSphere, 6.0, executes the same instruction sequence simultaneously in both VMs. This approach enables vSphere 6.0 to add support for up to 4 virtual CPUs in a protected VM.

Napper et al. [25] and Friedman and Kama [15] implemented fault-tolerant Java virtual machines using an approach similar to that of Bressoud and Schneider. This choice of hypervisor reduces overhead compared to virtual machine monitors that execute desktop operating systems, but also limits the class of applications.

Replication may be achieved by copying the state of a system instead of replaying input deterministically. State copying applies to multiprocessors and does not require control of non-determinism, but replicating state typically requires higher bandwidth than replicating inputs. In contrast to the replaying VM replication systems discussed above, Cully et al. [11] implemented state-copying primary-backup VM replication in Remus by building on live migration in the Xen virtual machine monitor [10]. Remus uses techniques such as pipelining execution with replication to address the high overheads of checkpointing VM state. Later work [24] established still further gains by compressing the replicated data, and other gains specific to paravirtualized systems. Tardigrade builds on the approach used in Remus and further reduces overhead.

Additional optimizations to VM replication have been proposed [23, 36, 37]. Although some optimizations are specific to a virtualization platform, others, such as speculative state transfer, may apply to Tardigrade and are topics for future work.

An LVM is similar in many regards to the containers actively being developed for Linux, including Docker [12] and its associated kernel support from LXC/LXD [21]. While there has been previous work on process-level migration in a research context [14, 28], the current popular interest in Linux containers makes us believe LVMR may be valuable outside the research community. Container interfaces appear to be closing the gap between the strong but efficient runtime isolation that is achieved by LVM and LXC/LXD and the desire to more easily deliver and manage the lifecycles of entire application stacks in production environments. There is already active open-source work on providing live container migration for Linux [8], and the work described in this paper is a natural next direction. Containers are actively used to manage large-scale distributed applications today, so integrating LVMR into those environments would ease the development complexity associated with critical, central components like those described in §6.1.

Designing services to be fault-tolerant

Virtualization-based fault tolerance is useful not only for protecting unmodified legacy applications, but also for reducing the cost and effort of developing new fault-tolerant services. In this section we overview some alternate, non-transparent approaches and their complexities.

One approach is to write the service as a serializable, deterministic state machine and rely on a library such as BFT [6] or SMART [22] for replication. However, there are several common errors the developer can make that will invisibly undermine the library's consistency guarantees, such as failing to serialize all relevant data, or writing non-deterministic code [1]. More recent work on Eve [19] has shown how to lift the requirement of determinism, but still requires annotation of which objects need serialization and, for performance, which operations are likely to commute.

Another approach is to micromanage the persistence of state to a reliable storage backend at the application level. One class of systems exemplifying this approach is transactional databases. Such customization is likely to yield good performance but requires careful engineering, including non-trivial checkpointing and recovery mechanisms [16]. A range of backend solutions are available, from locally-administered disk arrays [9] and distributed file systems to cloud-hosted services.

8 Conclusions

This paper describes asynchronous lightweight virtual machine replication, a technique for automatically converting an existing service into one that will tolerate machine failures. Using LVMs instead of VMs has the advantage that only changes to the application's state need to be replicated to backups before network output can be released. This leads to reasonable client-perceived latencies, even at the 99.9th quantile, and even when external processes share the host. To demonstrate the practicality of LVMR, we implemented it in the Tardigrade system. Tardigrade is not suitable for services that require extremely low latency or that modify memory at high rates. But, for many other services, the benefit of transparent fault tolerance will be a welcome aid to developers who lack the time, inclination, or expertise to correctly make their services consistent and fault tolerant.

9 Acknowledgments

The authors thank James Mickens and Jeremy Elson for helping us with FDS cluster configuration, Brian Zill and Jitu Padhye for answering networking questions, and Reuben Olinsky for help with Bascule. We are grateful to the anonymous reviewers for their helpful comments, and particularly to our shepherd, Jeff Chase, whose substantial feedback improved the paper.

References

[1] A. Adya, W. J. Bolosky, M. Castro, G. Cermak, R. Chaiken, J. R. Douceur, Jon, J. Howell, J. R. Lorch, M. Theimer, and R. P. Wattenhofer. FARSITE: Federated, available, and reliable storage for an incompletely trusted environment. In *Proc. OSDI*, 2002.

[2] C. Amza, A. L. Cox, S. Dwarkadas, P. Keleher, H. Lu, R. Rajamony, W. Yu, and W. Zwaenepoel. TreadMarks: Shared memory computing on networks of workstations. *Computer*, 29(2):18–28, 1996.

[3] A. Baumann, D. Lee, P. Fonseca, L. Glendenning, J. Lorch, B. Bond, R. Olinsky, and G. Hunt. Composing OS extensions safely and efficiently with Bascule. In *Proc. EuroSys '13*, pages 239–252, 2013.

[4] T. C. Bressoud and F. B. Schneider. Hypervisor-based fault tolerance. *ACM Transactions on Computer Systems (TOCS)*, 14(1):80–107, Feb. 1996.

[5] M. Burrows. The Chubby lock service for loosely-coupled distributed systems. In *Proc. OSDI*, 2006.

[6] M. Castro and B. Liskov. Practical Byzantine fault tolerance and proactive recovery. *ACM Transactions on Computer Systems (TOCS)*, 20(4):398–461, Nov. 2002.

[7] K. M. Chandy and L. Lamport. Distributed snapshots: Determining global states of distributed systems. *ACM Transactions on Computer Systems (TOCS)*, 3(1):63–75, Feb. 1985.

[8] Checkpoint/Restore in Userspace. http://criu.org/.

[9] P. M. Chen, E. K. Lee, G. A. Gibson, R. H. Katz, and D. A. Patterson. RAID: High-performance, reliable secondary storage. *ACM Computing Surveys (CSUR)*, 26(2):145–185, 1994.

[10] C. Clark, K. Fraser, S. Hand, J. G. Hansen, E. Jul, C. Limpach, I. Pratt, and A. Warfield. Live migration of virtual machines. In *Proc. NSDI*, pages 273–286. USENIX, 2005.

[11] B. Cully, G. Lefebvre, D. Meyer, M. Feeley, N. Hutchinson, and A. Warfield. Remus: High availability via asynchronous virtual machine replication. In *Proc. NSDI*, pages 161–174, 2008.

[12] Docker. https://www.docker.com/.

[13] J. R. Douceur, J. Elson, J. Howell, and J. R. Lorch. Leveraging legacy code to deploy desktop applications on the web. In *OSDI*, pages 339–354, Dec. 2008.

[14] F. Douglis and J. Ousterhout. Transparent process migration: Design alternatives and the Sprite implementation. *Softwware Practice and Experience*, 21(8):757–785, July 1991.

[15] R. Friedman and A. Kama. Transparent fault-tolerant Java virtual machine. In *Proc. Reliable Distributed Systems*, pages 319–328. IEEE, 2003.

[16] J. Gray, P. McJones, M. Blasgen, B. Lindsay, R. Lorie, T. Price, F. Putzolu, and I. Traiger. The recovery manager of the System R database manager. *ACM Computing Surveys (CSUR)*, 13(2):223–242, 1981.

[17] D. Gupta, S. Lee, M. Vrable, S. Savage, A. C. Snoeren, G. Varghese, G. M. Voelker, and A. Vahdat. Difference Engine: Harnessing memory redundancy in virtual machines. In *Proc. OSDI*, 2008.

[18] P. Hunt, M. Konar, F. P. Junqueira, and B. Reed. ZooKeeper: Wait-free coordination for Internet-scale systems. In *Proc. USENIX ATC*, 2010.

[19] M. Kapritsos, Y. Wang, V. Quema, A. Clement, L. Alvisi, and M. Dahlin. All about Eve: Execute-verify replication for multi-core servers. In *Proc. OSDI*, 2012.

[20] L. Lamport, D. Malkhi, and L. Zhou. Vertical Paxos and primary-backup replication. In *Proc. PODC*, 2009.

[21] Linux Containers. https://linuxcontainers.org/.

[22] J. R. Lorch, A. Adya, W. J. Bolosky, R. Chaiken, J. R. Douceur, and J. Howell. The SMART way to migrate replicated stateful services. In *Proc. EuroSys*, 2006.

[23] M. Lu and T. Chiueh. Fast memory state synchronization for virtualization-based fault tolerance. In *Proc. Dependable Systems Networks*, pages 534–543. IEEE, 2009.

[24] U. F. Minhas, S. Rajagopalan, B. Cully, A. Aboulnaga, K. Salem, and A. Warfield. RemusDB: Transparent high availability for database systems. *VLDB*, 22(1):29–45, 2013.

[25] J. Napper, L. Alvisi, and H. Vin. A fault-tolerant Java virtual machine. In *Proc. Dependable Systems and Networks (DSN)*, pages 425–425. IEEE Computer Society, 2003.

[26] E. B. Nightingale, P. M. Chen, and J. Flinn. Speculative execution in a distributed file system. In *Proc. SOSP*, 2005.

[27] E. B. Nightingale, J. Elson, J. Fan, O. Hofmann, J. Howell, and Y. Suzue. Flat datacenter storage. In *Proc. OSDI*, 2012.

[28] S. Osman, D. Subhraveti, G. Su, and J. Nieh. The design and implementation of Zap: A system for migrating computing environments. In *Proc. OSDI*, 2002.

[29] PaX Team. PaX address space layout randomization (ASLR). http://pax.grsecurity.net/docs/aslr.txt.

[30] D. Porter, S. Boyd-Wickizer, J. Howell, R. Olinsky, and G. Hunt. Rethinking the library OS from the top down. In *Proc. ASPLOS XVI*, pages 291–304, 2011.

[31] D. J. Scales, M. Nelson, and G. Venkitachalam. The design of a practical system for fault-tolerant virtual machines. *ACM SIGOPS Operating Systems Review*, 44(4):30–39, 2010.

[32] F. B. Schneider. Implementing fault-tolerant services using the state machine approach: A tutorial. *ACM Computing Surveys (CSUR)*, 22(4):299–319, 1990.

[33] C. A. Waldspurger. Memory resource management in VMware ESX Server. In *Proc. OSDI*, 2002.

[34] D. A. Wheeler. SLOCCount. http://www.dwheeler.com/sloccount/.

[35] O. Whitehouse. An analysis of address space layout randomization on Windows Vista. Technical report, Symantec, 2007.

[36] J. Zhu, W. Dong, Z. Jiang, X. Shi, Z. Xiao, and X. Li. Improving the performance of hypervisor-based fault tolerance. In *Symposium on Parallel & Distributed Processing (IPDPS)*, pages 1–10. IEEE, 2010.

[37] J. Zhu, Z. Jiang, Z. Xiao, and X. Li. Optimizing the performance of virtual machine synchronization for fault tolerance. *IEEE Transactions on Computers*, 60(12):1718–1729, 2011.

Retro: Targeted Resource Management in Multi-tenant Distributed Systems

Jonathan Mace[1], Peter Bodik[2], Rodrigo Fonseca[1], Madanlal Musuvathi[2]
[1]Brown University, [2]Microsoft Research

Abstract

In distributed systems shared by multiple tenants, effective resource management is an important pre-requisite to providing quality of service guarantees. Many systems deployed today lack performance isolation and experience contention, slowdown, and even outages caused by aggressive workloads or by improperly throttled maintenance tasks such as data replication. In this work we present Retro, a resource management framework for shared distributed systems. Retro monitors per-tenant resource usage both within and across distributed systems, and exposes this information to centralized resource management policies through a high-level API. A policy can shape the resources consumed by a tenant using Retro's control points, which enforce sharing and rate-limiting decisions. We demonstrate Retro through three policies providing bottleneck resource fairness, dominant resource fairness, and latency guarantees to high-priority tenants, and evaluate the system across five distributed systems: HBase, Yarn, MapReduce, HDFS, and Zookeeper. Our evaluation shows that Retro has low overhead, and achieves the policies' goals, accurately detecting contended resources, throttling tenants responsible for slowdown and overload, and fairly distributing the remaining cluster capacity.

1 Introduction

Most distributed systems today are *shared* by multiple tenants, both on private and public clouds and datacenters. These include common storage, data analytics, database, queuing, or coordination services like Azure Storage [6], Amazon SQS [3], HDFS [36], or Hive [41]. Multi-tenancy has clear advantages in terms of cost and elasticity.

However, providing performance guarantees and isolation in multi-tenant distributed systems is extremely hard. Tenants not only share fine-grained resources within a process (such as threadpools and locks) but also resources across multiple processes and machines (such as the disk and the network) along the execution path of their requests. As a result, traditional resource management mechanisms in the operating system and in the hypervisor are ineffective due to a mismatch in the management granularity. Moreover, tenant-generated requests not only compete with each other but also with system-generated tasks, such as replication and garbage collection, for shared resources. In addition, the bottleneck responsible for degrading the performance of a tenant can change in unpredictable ways depending on its input workload, the workload of other tenants and system tasks, the overall state of the system (including caches), and the (nonlinear) performance characteristics of underlying resources. See Figure 1 for an example. It does not help that the APIs to these services are often complex, with HDFS, for example, having over 100 calls in its client library [19], making static workload models intractable.

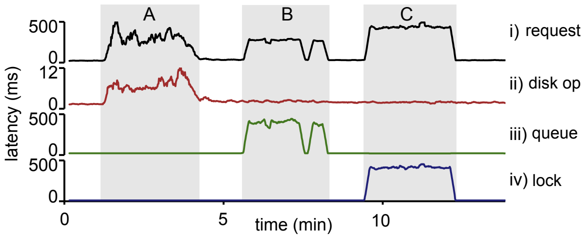


Figure 1: i) Latency for a client reading 8kB files from HDFS [36] is impacted by different workloads that A) replicate HDFS blocks, B) list large directories, and C) make new directories, each overloading the disk, threadpool, and locks respectively. ii) latency of DataNode disk operations, iii) latency at NameNode RPC queue, iv) latency to acquire NameNode "NameSystem" lock.

We address these challenges with Retro, a resource management framework whose core principle is to separate resource management policies from the mechanisms required to implement them. Retro enables system designers to state, verify, tune, and maintain management policies independent of the underlying system implementation. As in software defined networking, Retro policies execute in a *logically-centralized* controller with Retro mechanisms providing a global view of resource usage both within and across processes and machines.

The goal of Retro is to enable *targeted* policies that achieve desired performance guarantee or fairness goals by identifying and only throttling the tenants or system activities responsible for resource bottlenecks. Retro provides three abstractions to simplify the development of such policies. First, it groups all system activities – both tenant-generated requests and system-generated tasks – into individual *workflows*, which form the units of resource management. Retro attributes the usage of a resource at any instant to some workflow in the system. Second, Retro provides a *resource* abstraction that unifies arbitrary resources, such as physical storage, network, CPU, thread pools, and locks, enabling resource-agnostic policies. Each resource exposes two opaque performance metrics: *slowdown*, a measure of resource contention, and a per-workflow *load*, which attributes the resource usage to workflows. Finally, Retro creates *control points*, places in the system that implement resource scheduling mechanisms such as token buckets, fair schedulers, or priority queues. Each control point schedules requests locally, but is configured centrally by the policy.

Retro advocates *reactive* policies that dynamically respond to the current resource usage of workflows in the system, instead of relying on static models of future resource requirements. These policies continuously react to changes in resource bottlenecks and input workloads by making small adjustments directing the system towards a desired goal. Such a "hill climbing" approach enables policies that are robust to both changes in workload characteristics and nonlinear performance characteristics of underlying resources.

We evaluate Retro abstractions and design principles by implementing two fairness policies – reactive version of bottleneck resource fairness [12] and dominant resource fairness [13] – and LATENCYSLO, that enforces end-to-end latency targets for a subset of workflows. We use these policies on a Retro implementation for the Hadoop stack, comprising HDFS, Yarn, MapReduce, HBase and ZooKeeper. All three policies are concise (about 20 lines of code) and are agnostic of Hadoop internals. We experimentally demonstrate that these policies are robust and converge to desired performance goals for different types of workloads and bottlenecks.

The targeted and reactive policies of Retro rely on accurate, near real-time measurements of resource usage across all workflows and all resources in the system. Through a careful design of mostly-automatic instrumentation and aggregation of resource usage measurements our implementation of Retro for the Hadoop stack incurs latency and throughput overhead of 0.3% to 2%.

The goal of Retro is to be a general resource management framework that is applicable to arbitrary distributed systems. Our experience applying Retro to five distributed systems – HDFS, Yarn, MapReduce, HBase, and ZooKeeper – validates our design. Applying Retro to a new system required modest amounts of system-specific instrumentation – between 50 and 200 lines of code. The rest of the Retro framework required no changes. Moreover, resource management policies that we originally developed for HDFS were directly applicable to other systems, validating the robustness of Retro abstractions.

In summary, our key contributions are:

- Unifying abstractions of *workflows*, *resources*, and *control points*, that enable concise policies that are system-agnostic and resource-agnostic;
- Demonstrating the feasibility of Retro in a complex Hadoop stack, including a low-overhead, pervasive, per-workflow resource tracking and aggregation for a wide variety of resources;
- Targeted and reactive policies for providing latency SLOs, bottleneck resource fairness, and dominant resource fairness;
- A centralized controller that allow policies to enforce performance goals at different control points without requiring explicit coordination.

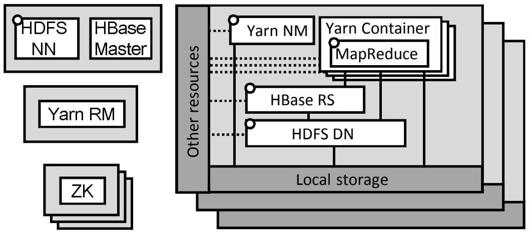


Figure 2: Typical deployment of HDFS, ZooKeeper, Yarn, MapReduce, and HBase in a cluster. Gray rectangles represent servers, white rectangles are processes, and white circles represent control points that we added. See text for details.

2 Motivation and challenges

This section motivates Retro by describing the challenges of resource management in a multi-tenant distributed system. As this paper presents Retro in the context of the Hadoop stack, we first provide a high-level overview of Hadoop components. The results of this paper generalize to other distributed systems as well.

2.1 Hadoop architecture

Figure 2 shows the relevant components of the Hadoop stack. HDFS [36], the distributed file system, consists of DataNodes (DN) that store replicated file blocks and run on each worker machine, and a NameNode (NN) that manages the filesystem metadata. Yarn [42] comprises a single ResourceManager (RM), which communicates with NodeManager (NM) processes on each worker. Hadoop MapReduce is an application of Yarn that runs its processes (application master and map and reduce tasks) inside Yarn containers managed by NMs. HBase [17] is a data store running on top of HDFS that consists of RegionServers (RS) on all workers and an HBase Master, potentially co-located with the NameNode or Yarn. Finally, ZooKeeper [21] is a system for distributed coordination used by HBase.

MapReduce job input and output files are loaded from HDFS or HBase, but during the job's *shuffle* phase, intermediate output is written to local disk by mappers (bypassing HDFS) and then read and transferred by NodeManagers to reducers. Reading and writing to HDFS has the

NameNode on the critical path to obtain block metadata. An HBase query executes on a particular RegionServer and reads/writes its data from one or many DataNodes.

2.2 Resource management challenges

Any resource can become a bottleneck Figure 1 demonstrates how the latency of an HDFS client can be adversely affected by other clients executing very different types of requests, contending for different resources. In production, a Hadoop job that reads many small files can stress the storage system with disk seeks, as workload A in the figure, and impact all other workloads using the disks. Similarly, a workload that repeatedly resubmits a job that fails quickly puts a large load on the NN, like workload C, as it has to list all the files in the job input directories. In communication with Cloudera [43], they acknowledge several instances of aggressive tenants impacting the whole cluster, saying "anything you can imagine has probably been done by a user". Interviews with service operators at Microsoft confirm this observation.

Multiple granularities of resource sharing On the one hand, concurrently executing workflows share software resources, such as threadpools and locks, within a process, while on the other hand, resources, such as the disk on Hadoop worker nodes, are distributed across the system. The disk resource, for example, is accessed by DN, NM, and mapper/reducer processes running across all workers. Systems have many entry points (*e.g.*, HBase, HDFS, or MapReduce API) and maintenance tasks are launched from inside the system. Finally, enforcing resource usage for long-running requests requires throttling inside the system, not just at the entry points.

Maintenance and failure recovery cause congestion Many distributed systems perform background tasks that are not directly triggered by tenant requests but compete for the same resources. E.g., HDFS performs data replication after failures, asynchronous garbage collection after file deletion, and block movement for balancing DN load. In some cases, these background tasks can adversely affect the performance of foreground tasks. Jira HDFS-4183 [18] describes an example where a large number of files are abandoned without closing, triggering a storm of block recovery operations after the lease expiration interval one hour later, which overloads the NN. Guo et al. [16] describe a failure in Microsoft's datacenter where a background task spawned a large number of threads, overloading the servers. On the other hand, some of these tasks need to be protected *from* foreground tasks. Guo et al. [16] describe a cascading failure resulting from overloaded servers not responding to heartbeats, triggering further data replication and further overload.

Resource management is nonexistent or noncomprehensive Systems like HDFS, ZooKeeper, and HBase do not contain any admission control policies. While Yarn allocates compute slots using a fair scheduler, it ignores network and disk, thus, an aggressive job can overload these resources. Interviews with service operators at Microsoft indicate that productions system often implement resource management policies that ignore important resources and use hardcoded thresholds. For example, a policy might assume that an `open()` is 2x more expensive than `delete()`, while the actual usage varies widely based on parameters and system state, resulting in very inaccurate resource accounting. The policies are often tweaked manually, typically *after* causing performance issues or outages, or when the system or the workloads change. Writing the policies often requires intimate knowledge of the system and of the request resource profile, which may be impossible to know a priori.

3 Design

The main goal of Retro is to enable simple, targeted, system-agnostic, and resource-agnostic resource-management polices for multi-tenant distributed systems. Examples of such policies are: a) throttle aggressive tenants who are getting an unfair share of bottlenecked resources, b) shape workflows to provide end-to-end latency or throughput guarantees, or c) adjust resource allocation to either speed up or slow down certain maintenance or failure recovery tasks.

Retro addresses the challenges in §2.2 by separating the mechanisms of measurement and enforcement of resource usage from high-level, global resource management policies. It does this by using three unifying abstractions – *workflows*, *resources*, and *control points* – that enable logically centralized policies to be succinctly expressed and apply to a broad class of resources and systems.

3.1 Retro abstractions

Workflow Resource contention in a distributed system can be caused by a wide range of system activities. Retro treats each such activity as a first-class entity called a *workflow*. A workflow is a set of requests that forms the unit of resource measurement, attribution, and enforcement in Retro. For instance, a workflow might represent requests from the same user, various background activities (such as heartbeats, garbage collection, or data load balancing operations), or failure recovery operations (such as data replication). The aggregation of requests into a workflow is up to the system designer. For instance, one system might treat all background activities as one workflow but another might treat heartbeats as a distinct workflow from other activities, if the system designer decides to provide a different priority to heartbeats.

Each workflow has a unique workflow ID. To properly attribute resource usage to individual workflows, Retro propagates the workflow ID along the execution path of all requests. This causal propagation [11, 37, 40, 34] allows

Retro to attribute the usage of a resource to a workflow at any point in the execution, whether within a shared process or across the network.

Resources A comprehensive resource management policy should be able to respond to a contention in any resource – hardware or software – and attribute load to workflows using it. A key hypothesis of Retro is that resource management policies can and should treat all resources, from thread pools to locks to disk, uniformly under a common abstraction. Such a uniform-treatment allows one to state policies that respond to disk contention, say, in the same way as lock contention. Equally importantly, this allows gradually expanding the scope of resource-management to new resources without policy change. For instance, a storage service might start by throttling clients based on their network or disk usage. However, as the complexity of the service increases to include sophisticated meta-data operations, the service can start throttling by CPU usage or lock-contention. On the other hand, the challenge in providing such a unifying abstraction is to capture the behavior of varied kinds of resources with different complex non-linear performance characteristics.

To overcome this challenge, Retro captures a resource's current *first-order* performance with two unitless metrics:

- **slowdown** indicates how slow the resource is currently, compared to its baseline performance with no contention;
- **load** is a per-workflow metric that determines who is responsible for the slowdown.

As a simple example, consider an abstract resource with an (unbounded) queue. Let $Q_{w,i}$ be the queueing time of the ith request from workflow w in a time interval and let $S_{w,i}$ be the time the resource takes to service that request. During this interval, the load by w is $\Sigma_i S_{w,i}$ and the slowdown is $\Sigma_{w,i}(Q_{w,i} + S_{w,i})/\Sigma_{w,i}S_{w,i}$. Note, the denominator of the slowdown is the time taken to process the requests if the queue is empty throughout the interval.

The reactive policies in Retro allow these metrics to provide a linear approximation of the complex non-linear behavior. The policies continuously measure the resource metrics while making incremental resource allocation changes. Operating in such a feedback loop enables simple abstractions while reacting to nonlinearities in the underlying performance characteristics of the resource.

Resources in real systems are more complex than the simple queue above. Our goal is to hide the complexities of measuring the load and slowdown of different resources in *resource libraries* that are implemented once and reused across systems. See §4.2 for details.

An important implication of this abstraction is that it is not possible to query the capacity of a resource. Instead, a policy can treat a resource to have reached its capacity if the slowdown exceeds some fixed constant. Directly measuring true capacity is often not possible because of many request types supported (*e.g.* open, read, sync, etc. on a disk) and because of effects of caching or buffering, workflow demands do not compose linearly. Also, due to limping hardware [10], estimating the current operating capacity is next to impossible.

Control points To separate the low-level complexities of enforcing resource allocation throughout the distributed system, we introduce the *control point* abstraction. A control point is a point in the execution of a request where Retro can enforce the decisions of resource scheduling policies. Each control point executes locally, such as delaying requests of a workflow using a token bucket, but is configured centrally from a policy.

While a control point can be placed directly in front of a resource (such as a thread pool queue), it can more generally be located anywhere it is reasonable to sleep threads or delay requests, such as in HDFS threads sending and receiving data blocks. The location of control points should be selected by the system designer while keeping a few rules in mind. A control point should not be inserted where delaying a request can directly impact other workflows, such as when holding an exclusive lock. Conversely, some asynchronous design patterns (such as thread pools) present an opportunity to interpose control points, as it is unlikely that a request will hold critical resources yet potentially block for a long period of time.

Each logical control point has one or more instances. A point with a single instance is centralized, such as a point in front of the RPC queue in HDFS NameNode. Distributed points, such as in the DataNode or its clients, have many, potentially thousands of instances. Each instance measures the current, per-workflow throughput which is aggregated inside the controller.

To achieve fine-grained control, a request has to periodically pass through control points, otherwise, it could consume unbounded amount of resources. To illustrate this, consider a request in HBase that scans a large region, reading data from multiple store files in HDFS. If Retro only throttles the request at the RegionServer RPC queue, a policy has only one chance to stop the request; once it enters HBase, it can read an unbounded amount of data from HDFS and perform computationally expensive filters on the data server-side. By adding a point to the DataNode block sender, we can control the workflow at the granularity of 64kB HDFS data packets. More generally, the longer the period of time a request can execute without passing through a control point, the longer it will take any policy to react. This is similar to the dependence between the longest packet length L_{max} and the fairness guarantees provided by packet schedulers [31, 38].

3.2 Architecture

Figure 3 outlines the high-level architecture of Retro and its three main components. First, Retro has a measure-

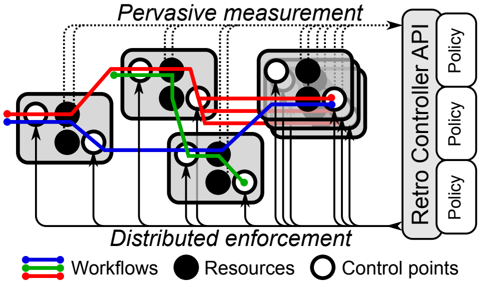


Figure 3: Retro architecture. Gray boxes are system components on the same or different machines. Workflows start at several points and reach multiple components. Intercepted resources (●) generate measurements that serve as inputs to policies. Policy decisions are enforced by control points (○).

ment infrastructure that provides near-real-time resource usage information across all system resources and components, segmented by workload. Second, the logically centralized controller uses the resource library to translate raw measurements to the load and slowdown metrics, and provides them as input to Retro policies. Third, Retro has a distributed, coordinated enforcement mechanism that consistently applies the decisions of the policies to control points in the system. We discuss the design of the controller in the following paragraphs. In §4 we describe the measurement and enforcement mechanisms in detail, and in §5 we present the implementation of three policies.

Logically centralized policies In current systems, resource management policies are hard-coded into the system implementation making it difficult to maintain as the system and policies evolve. A key design principle behind Retro is to separate the mechanisms (§4) from the policies (§5). Apart from making such policies easier to maintain, such a separation allows policies to be reused across different systems or extended with more resources.

Borrowing from the design of Software Defined Networks and IOFlow [39], Retro takes the separation a step further by logically centralizing its policies. This makes policies much easier to write and understand, as one does not have to worry about myopic local policies making conflicting decisions. In this light, we can view Retro as building a "control plane" for distributed systems, and providing a separation of concerns for policy writers and system developers and instrumenters.

Retro exposes to policies a simple API, shown in Table 1, that abstracts the complexity of individual resources and allows one to specify resource-agnostic scheduling policies, as demonstrated in §5. The first three functions in the table correspond to the three abstractions explained above. In addition, `latency(r,w)` returns the total time workflow `w` spent using resource `r`. `throughput(p,w)` measures the aggregate request rate of workflow `w` through a (potentially distributed) throttling point `p`, such as the entry point to the RS process. Finally, policies can affect the system through Retro's throttling mechanisms.

`workflows()`	list of workflows
`resources()`	list of resources
`points()`	list of throttling points
`load(r,w)`	load on `r` by workflow `w`
`slowdown(r)`	slowdown of resource `r`
`latency(r,w)`	total latency spent by `w` on `r`
`throughput(p,w)`	throughput of workflow `w` at point `p`
`get_rate(p,w)`	get the throttling rate of workflow `w` at point `p`
`set_rate(p,w,v)`	throttle workflow `w` at point `p` to `v`

Table 1: Retro API used by the scheduling policies. We omit auxiliary calls to set, for example, the reporting interval and smoothing parameters, as well as to obtain more details such as operation counts, etc.

4 Implementation

4.1 Per-workflow resource measurement

End-to-end ID propagation At the beginning of a request, Retro associates threads executing the request with the workflow by storing its ID in a thread local variable; when execution completes, Retro removes this association. While the developer has to manually propagate the workflow ID across RPCs or in batch operations, we use AspectJ to automatically propagate the workflow ID when using `Runnable`, `Callable`, `Thread`, or a `Queue`.

Aggregation and reporting When a resource is intercepted, Retro determines the workflow associated with the current thread, and increments in-memory counters that track the per-workflow resource use. These counters include the number of resource operations started and ended, total latency spent executing in the resource and any operation-specific statistics such as bytes read or queue time. When the workflow ID is not available, such as when parsing an RPC message from the network, the resource use is attributed to the *next* ID that is set on the current thread (*e.g.*, after extracting the workflow ID from the RPC message). Retro does not log or transmit individual trace events like X-Trace or Dapper, but only aggregates counters in memory. A separate thread reads and reports the values of the counters to the central controller at a fixed interval, currently once per second. Reports are serialized using protocol buffers [14] and sent using ZeroMQ [2] pub-sub. The centralized controller aggregates reports by workflow ID and resource, smoothes out the values using exponential running average, and uses the resource library to compute resource load and slowdown.

Batching In some circumstances, a system might batch the requests of multiple workflows into a single request. HDFS NameNode, HBase RegionServers, and ZooKeeper each have a shared transaction log on the critical path of write requests. In these cases, we create a *batch workflow ID* to aggregate resource consumption of the batch task (e.g., the resources consumed when writing HBase transaction logs to HDFS). Constituent workflows report their relative contributions to the batch (e.g., serialized size of transaction) and the controller decomposes the resources consumed by the batch to the contributing workflows.

Automatic resource instrumentation using AspectJ Retro uses AspectJ [25] to automatically instrument all hardware resources and resources exposed through the Java standard library. Disk and network consumption is captured by intercepting constructor and method calls on file and network streams. CPU consumption is tracked during the time a thread is associated with a workflow. Locking is instrumented for all Java monitor locks and all implementers of the `Lock` interface, while thread pools are instrumented using Java's `Executors` framework. The only manual instrumentation required is for *application-level* resources created by the developer, such as custom queues, thread pools, or pipeline processing stages.

AspectJ is highly optimized and *weaves* the instrumentation with the source code when necessary without additional overheads. In order to avoid potentially expensive runtime checks to resolve virtual function calls, Retro instrumentation only intercepts constructors to return proxy objects that have instrumentation in place.

4.2 Resource library

Retro presents a unified framework that incorporates individual models for each type of resource. Management policies only make incremental changes to request rates allocated to individual workflows; for example, if the CPU is overloaded, a policy might reduce total load on the CPU by 5%. Therefore, as long as we correctly detect contention on a resource, iteratively reducing load on that resource will reduce the contention. Our models, thus, capture only the first-order impact of load on resource slowdown.

CPU We query the per-thread CPU cycle counter when setting and unsetting the workflow ID on a thread (using `QueryThreadCycleTime` in Windows and `clock_gettime` in Linux) to count the total number of CPU cycles spent by each workflow. The load of a workflow is thus proportional to its usage of CPU cycles. To estimate the slowdown, we divide the actual latency spent using CPU by the *optimal latency* of executing this many cycles at the CPU frequency. Since part of the thread execution could be spent in synchronous IO operations, we only use CPU cycles and latency spent outside of these calls to compute CPU slowdown. If frequency scaling is enabled, we could use other existing performance counters to detect CPU contention [1].

Disk To estimate disk slowdown, we use a subset of disk IO operation types that we monitor, in particular, `reads` and `syncs`. For example, given a time interval with n `syncs` and b bytes written during these operations, we use a simple disk model that assumes a single seek with duration T_s for each `sync`, followed by data transfer at full disk bandwidth B. We thus estimate the optimal latency as $l = nT_s + b/B$ and slowdown as $s = t/l$, where t is the total time spent in `sync` operations. To deal with disk caching, buffering, and readahead, we only count as seeks the operations that took longer than a certain threshold, *e.g.*, 5ms. We use similar logic for reads and to estimate the load of each workflow.

Network The load of a workflow on a network link is proportional to the number of bytes transferred by that workflow. We ignore data sent over the loopback interface by checking remote address when the connection is set up, inside our AspectJ instrumentation. We currently do not measure the actual network latency and thus estimate the network slowdown based on its utilization by treating it as a M/M/1 queue. Thus a link with utilization u has a slowdown $1 + u/(1-u)$. It is feasible to extend Retro by encoding a model of the network (topology, bandwidths, and round trip times), and network flow parameters (source, destination, number of bytes), to estimate the network flow latency with no congestion [30]. Comparing this no-congestion estimate with measured latency could be used to compute network slowdown.

Thread pool The load of a workflow on a thread pool is proportional to the total amount of time it was using threads in this pool. Since we explicitly measure queuing and service time of a thread pool operation, we can directly compute the slowdown as total execution time (queuing plus service) divided by service time.

Locks A write lock behaves similarly to a thread pool with a single thread, and we explicitly measure the queuing time of a lock operation and the time the thread was holding the lock. Slowdown is thus the total latency of lock operation (from requesting the lock until release) divided by the time actually holding the lock.

Load of a read-write lock depends on the number of read and write operations, for how long they hold the lock, and the exact lock implementation. While there has been previous work on modeling locks using queues [24, 22, 32], none of them exactly match the ReentrantReadWriteLock used in HDFS. Instead, we approximate the capacity or throughput of a lock, $T(f,w,r)$, in a simple benchmark using three workflow parameters: fraction of write locks f, and average duration of write and read locks w and r. See Figure 4 for a subset of the measured values; notice that the throughput is nonlinear and non-monotonic. We use trilinear interpolation [23] to predict throughput for values not directly measured. Given a workflow with characteristic (f,w,r) and current lock throughput t, we estimate its load on the lock as

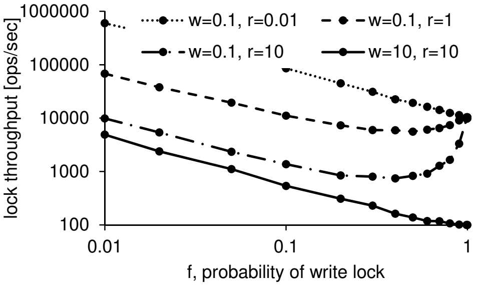


Figure 4: The throughput of Java ReentrantReadWriteLock (y-axis) as a function of three parameters: probability of a write lock operation (x-axis), average duration of read and write locks (see legend, time in milliseconds).

$t/T(f,r,w)$. E.g., a workflow making 1000 lock requests a second with its estimated max throughput of 5000 operations a second, would have a load of 0.2.

4.3 Coordinated throttling

Retro is designed to support multiple scheduling schemes, such as various queue schedulers or priority queues. In the current implementation of Retro, each control point is a *per-workflow distributed token bucket*. Threads can request tokens from the current workflow's token bucket, blocking until available. Queues can delay a request from being dequeued until sufficient tokens are available in the corresponding workflow's bucket. For a particular control point and workflow, a policy can set a rate limit R, which is then split (behind the scenes) across all point instances proportionally to the observed throughput. Retro keeps track of new control point instances coming and going – *e.g.*, mappers starting and finishing – and properly distributes the specified limit across them.

So long as each request executes a bounded amount of work, even using a single control point at the entrance to the system is enough for Retro to enforce usage of individual workflows. However, as described in Section 3.1, requests have to periodically pass through control points to guarantee fast convergence of allocation policies. Even without any control points in the system, each resource reports how many times it has been used by a particular workflow. For example, loading a single HDFS block of 64MB would result in approximately 1000 requests to the disk, each reading 64kB of data. These statistics help developers identify blocks of code where requests execute large amount of work and where adding control points helps break down execution and significantly improves convergence of control policies.

In the Hadoop stack, we added several points: in the HDFS NameNode and HBase RegionServer RPC queues, in the HDFS DataNode block sender and receiver, in the Yarn NodeManager, and in the MapReduce mappers when writing to the local disk. Each of these points has a number of instances equal to the number of processes of the particular type.

```
1  // identify slowest resource
2  S = r in resources() with max slowdown(r)
3  foreach w in workflows()
4    demand[w] = load(S, w)
5    capacity += $(1-\alpha)$*demand[w]
6
7  fair = MaxMinFairness(capacity, demand)
8
9  foreach w in workflows()
10   if (slowdown(S) > T
11       && fair[w] < demand[w]) // throttle
12     factor = fair[w] / demand[w]
13   else // probe for more demand
14     factor = (1 + $\beta$)
15
16   foreach p in points()
17     set_rate(p, w, factor*get_rate(p, w))
```

Algorithm 1: BFAIR policy, see §5.1.

Notice that we do not need to throttle directly on resource R to enforce resource limits on R. Assume that a workflow is achieving throughput of N_p at point p and has load L_R on R. By setting a throttling rate of αN_p for all points, we will indirectly control the load on R to αL_R.

5 Policies

This section describes three targeted reactive resource-management policies that we used to evaluate Retro. Specifically, these policies enforce fairness on the bottleneck resource (§5.1), dominant-resource fairness (§5.2), and end-to-end latency SLOs (§5.3). All of these polices are system-agnostic, resource-agnostic, and can be concisely stated in a few lines of code. These are not the only policies that could be implemented on top of Retro; in fact, we believe that the Retro abstractions allow developers to write more complex policies that consider a combination of fairness and latency, together with other metrics, such as throughput, workflow priorities, or deadlines.

5.1 BFAIR policy

The BFAIR policy provides bottleneck fairness [13, 12]; *i.e.*, if a resource is overloaded, the policy reduces the total load on this resource while ensuring max-min fairness for workflows that use this resource. This policy can be used to throttle aggressive workflows or to provide DoS protection. It provides coarse-grained performance isolation, since workflows are guaranteed a fair-share of the bottlenecked resource.

The policy, described in Algorithm 1, first identifies the slowest resource S in the system according to the slowdown measure (line 2). Then, the policy runs the max-min fairness algorithm with demands estimated by the current load of workflows (line 4) and resource capacity estimated by the total demand reduced by $1-\alpha$ to relieve the bottleneck if any (line 5).

The policy considers S to be bottlenecked if its slowdown is greater than a policy-specific threshold T. If this is the case and the fair share fair[w] of workflow w is

```
1   // estimate resource demands
2   foreach w in workflows()
3     foreach r in resources()
4       demand[r,w] = $(1+\alpha)$*load(r,w)
5
6   // update capacity estimates
7   cap = current capacity estimates
8   foreach r in resources()
9     tot_load = $\Sigma_w$load(r,w)
10    if(slowdown(r) > $T_r$) //reduce estimate
11       cap[r] = min(cap[r], tot_load);
12    else // probe for more capacity
13       cap[r] = max((1+$\beta$)*cap[r], tot_load);
14
15  share = DRF(demand, cap)
16  foreach w in workflows()
17    if (share[w] >= 1) continue
18    foreach p in points()
19      set_rate(p, w, share[w]*get_rate(p,w))
```

Algorithm 2: RDRF policy, see §5.2.

```
1   foreach w in H
2     miss(w) = latency(w) / target_lat(w)
3   h = w in H with max miss(w)
4
5   foreach l in L // compute gradients
6     g[l] = $\Sigma_r$ (latency(h,r) * log(slowdown(r))
7                * load(r,l) / $\Sigma_w$ load(r,w))
8
9   foreach l in L // normalize gradients
10    g[l] /= $\Sigma_k$g[k]
11
12  foreach l in L
13    if(miss(h) > 1) // throttle
14      factor = 1-$\alpha$*(miss(h)-1)*g[l]
15    else // relax
16      factor = (1 + $\beta$)
17
18    foreach p in points()
19      set_rate(p, l, factor*get_rate(p, l))
```

Algorithm 3: LatencySLO policy, see §5.3.

smaller than its current load (line 11), the policy throttles the rate by a factor of fair[w]/demand[w]. Here, the policy assumes a linear relationship between throughput at control points and the load on resources. If either the resource is not bottlenecked or if a workflow is not meeting its fair share (line 13), the policy increases the throttling rate by a factor of $1+\beta$ to probe for more demand.

Notice that this policy performs *coordinated* throttling of the workflow across all the control points; by reducing the rate proportionally on each point, we quickly reduce the load of the workflow on all resources. Parameters α and β control how aggressively the policy reacts to overloaded resources and underutilized workflows respectively. Notice that this policy will throttle only if there is a bottleneck in the system; we can change the definition of a bottleneck using the parameter T.

5.2 RDRF policy

Dominant resource fairness (DRF) [13] is a multi-resource fairness algorithm with many desirable properties. The RDRF policy (Algorithm 2) calls the original DRF function at line 15 which requires the current resource demands and capacities of all resources. In a general distributed system, we cannot directly measure the *actual resource demand* of a workflow, but only its current load on a resource. A workflow might not be able to meet its demand due to bottlenecks in the system.

The RDRF policy overcomes this problem by being reactive: making incremental changes and reacting to how the system responds to these changes. At any instant, the policy conservatively assumes that each workflow can increase its current demand by a factor of α (line 4). This increased allocation provides room for bottlenecked workflows to increase the load on resources.

Similarly, the policy uses the slowdown measure to estimate capacity. At line 10, when the current slowdown exceeds a resource-specific threshold, the policy reduces its capacity estimate to the current load. On the other hand, if the slowdown is within the threshold (line 12) and the current capacity estimate is lower than the current load, the policy increases the capacity estimate by a factor of β to probe for more capacity.

Given estimates of demand and capacity, the DRF() function returns share[w], the fraction of w's demand that was allocated based on dominant-resource fairness. If share[w] < 1, we throttle w at each point p proportionally to its current throughput at p.

5.3 LatencySLO policy

In the LatencySLO policy, we have a set of high-priority workflows H with a specified target latency SLO (service-level objective). Let L (low-priority) be the remaining workflows. The goal of the policy is to achieve the highest throughput for L, while meeting the latency targets for H. We assume the system has enough capacity to meet the SLOs for H in the absence of the workflows L; in other words, it is not necessary to throttle H. To maximize throughput, we want to throttle workflows in L as little as possible; *e.g.*, if a workflow in L is not using an overloaded resource, it should not be throttled.

Consider a workflow h in H that is missing its target latency. If multiple such workflows exist, the policy choses the one with the maximum miss ratio (line 3). Let t_w be the current request rate of workflow w and consider a possible change of this rate to $t_w * f_w$. The resulting latency l_h of h is some (nonlinear) function of the relative workflow rates f_w of all workflows. The LatencySLO computes an approximate gradient of l_h with respect to f_w and uses the gradient to move the throttling rates in the right direction. Based on the system response, the policy repeats this process until all latency targets are met.

We derive an approximation of l_h which results in an intuitive throttling policy. Consider a resource r with a current slowdown of S_r, load $D_{w,r}$ for workflow w, and total load $D_r = \sum_w D_{w,r}$. If $L_{h,r}$ is the current latency of h at r, the baseline latency is $L_{h,r}/S_r$ when there is no load at r, by the definition of slowdown. We model the latency of h at r, $l_{h,r}$ as an exponential function of the load d_r that satisfies the current ($d_r = D_r$) and baseline ($d_r = 0$) latencies, and obtain $l_{h,r} = L_{h,r} * S_r^{d_r/D_r - 1}$. Finally, we model the latency of h, $l_h = \sum_r l_{h,r}$ as the sum of latencies across all resources in the system.

Assuming that a fractional change in a workflow's request rate results in the same fractional change in its load on the resources, we have $d_r = \sum_w D_{w,r} * f_w$. The gradient of $l_{h,r}$ with respect to f_w at $d_r = D_r$ is $\partial l_{h,r}/\partial f_w = L_{h,r} * \log S_r * D_{w,r}/D_r$. This is a very intuitive result: the impact of workflow w on the latency of h is high if it has a high resource share, $D_{w,r}/D_r$, on a resource with high slowdown, $\log S_r$, and where workflow h spends a lot of time, $L_{h,r}$.

Algorithm 3 uses this formula for the gradient calculation (line 6). The policy throttles workflows in L based on the normalized gradients after dampening by a factor α to ensure that the policy only takes small steps. If all workflows in H meet their latency guarantees, the policy uses this opportunity to relax the throttling by a factor β.

6 Evaluation

In this section we evaluate Retro in the context of the Hadoop stack. We have instrumented five open-source systems – HDFS, Yarn, MapReduce, HBase, and ZooKeeper – that are widely used in production today. We use a wide variety of workflows, which are based on real-world traces, widely-used benchmarks, and other workloads known to cause resource overload in production systems.

Our evaluation shows that Retro addresses the challenges in §2.2 when applied simultaneously to all these stack components. In particular, we show that Retro:

- applies coordinated throttling to achieve bottleneck and dominant resource fairness (§6.1 and §6.3);
- applies policies to application-level resources, resources shared between multiple processes, and resources with multiple instances across the cluster;
- guarantees end-to-end latency in the face of workloads contending on different resources, uniformly for client and system maintenance workflows (§6.2);
- is scalable and has very low developer and execution overhead (§6.4);
- throttles efficiently: it correctly detects bottlenecked resources and applies *targeted throttling* to the relevant workflows and control points.

We do not directly compare to other policies, since to our knowledge, no previous systems offer this rich source of per-workflow and per-resource data. Many of previous policies, such as Cake [44], could be directly implemented on top of Retro.

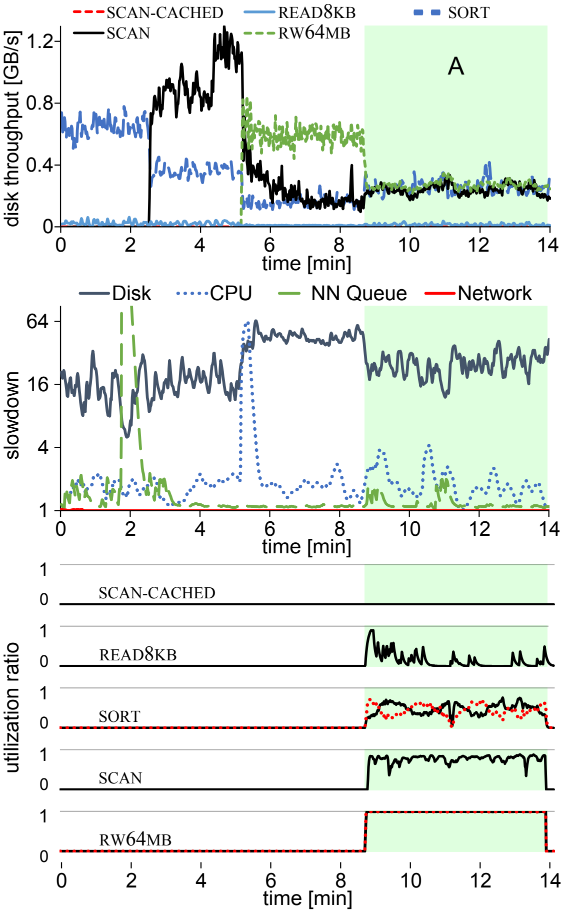


Figure 5: BFAIR policy as described in the text.

6.1 BFAIR in Hadoop stack

In Figure 5, we demonstrate the BFAIR policy successfully throttling aggressive workflows without negatively affecting the throughput of other workflows. The three *major* workflows are: SORT, a MapReduce sort job; RW64MB, 100 HDFS clients reading and writing 64MB files with a 50/50 split; and SCAN, 100 HBase clients scanning large tables. These workflows bottleneck on the disk on the worker machines. The two *minor* workflows are: READ8KB, 32 clients reading 8kB files from HDFS; and SCAN-CACHED, 32 clients scanning tables in HBase that are mostly *cached* in the RegionServers. We perform this experiment on a 32-node deployment of Windows Azure virtual machines; one node runs the Retro controller, one node runs HDFS NameNode, Yarn, ZooKeeper, and HBase RegionServer, the other thirty are used as Hadoop workers. Each VM is a Standard_A4 instance with 8 cores, 14GB RAM and a 600GB data disk, connected by a 1Gbps network.

At the beginning of the experiment, we start READ8KB, SCAN-CACHED, and SORT together, and delay start of SCAN and RW64MB. Figure 5(top) shows the disk throughput achieved by each workflow; notice how the throughput changes as different workflows start, for ex-

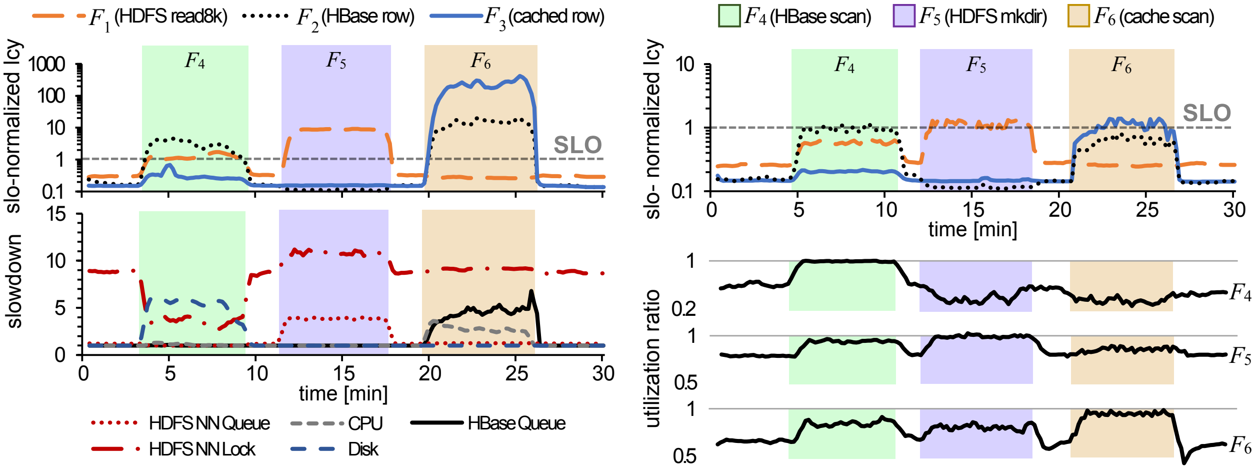


Figure 6: LATENCYSLO policy as described in text. Top-left figure shows high priority workflow latencies without LATENCYSLO. Bottom-left figure shows resource slowdown during experiment. Top-right figure shows high priority workflow latencies with LATENCYSLO. Bottom-right sparklines show control point utilizations for background workflows.

ample, throughput of SORT drops from 750MB/sec to 100MB/sec. Figure 5(center) shows the slowdown of a few different resources. Disk is the only constantly overloaded resource, reaching slowdown of up to 60. While slowdown of other resources also occasionally spikes, this happens only due to workload burstiness. In Figure 5(bottom), we show sparklines of the workflow *utilization ratios* – the achieved throughput relative to the allocated rate at a particular control point. A ratio of 1 means that the workflow is being actively rate-limited; a ratio of 0 means that the workflow is never rate-limited. For SORT, we show ratios at two control points: the DN BlockSender (black, used by mapper to read data from the DN) and mapper output (dashed red, used by mapper to write its output to local disk). For RW64MB, we show ratios at two control points: the DN BlockSender (black, used to read data from HDFS DNs) and the DN BlockReceiver (dashed red, used to write data to HDFS DNs).

In phase A we enable the BFAIR with overload threshold T=25. Quickly, the disk throughput of the three major workflows equalizes at about 300MB/sec, thus achieving fairness on the bottlenecked resource. Also, the disk slowdown fluctuates at around 25 (navy blue line in the slowdown graph) because the policy starts throttling the major workflows.

The utilization ratio sparklines provide further insight. RW64MB is the most aggressive workflow and consequently it is fully throttled (ratio of 1) at all of the control points. While not as aggressive, SCAN is also throttled though less. Depending on the phase of the map-reduce computation, we throttle SORT while reading input (black) and/or when writing output (red dashed). Finally, as expected, the two minor workflows are not throttled as much, or at all, because the fairness allocates their full demand. Furthermore, SCAN-CACHED is completely unthrottled as it has no disk utilization.

These results highlight how Retro enables coordinated and targeted throttling of workloads. No other system we are aware of would achieve these results, as Retro coordinates the same resource through different control points – for example, disk is controlled not only by HDFS block transfer (used by SCAN, RW64MB, READ8KB and the job input to SORT), but also by the SORT mapper output that accesses disk directly, bypassing HDFS. Retro only throttles the relevant workloads, leaving the small read and scan workloads mostly alone.

6.2 LATENCYSLO

We demonstrate that the LATENCYSLO policy can enforce a) end-to-end latency SLOs across multiple workflows and systems, and b) SLOs for both front-end clients and background tasks. We perform these experiments on an 8-node cluster; one node runs the Retro controller, one node runs HDFS NameNode, Yarn, ZK, and HBase Master, the other 6 are used as Hadoop workers and HBase RegionServers.

Enforcing multiple guarantees In this experiment we simultaneously enforce SLOs in HBase and HDFS for three high priority workflows with intermittently aggressive background workflows. The three high priority workflows are: F_1 randomly reads 8kB from HDFS with 500ms SLO, F_2 randomly reads 1 row from a small table cached by HBase with 25ms SLO, and F_3 randomly reads 1 row from a large HBase table with 250ms SLO. The background workflows are: F_4 submits 400-row HBase table scans, F_5 creates directories in HDFS, and F_6 submits 400-row HBase table scans of a cached HBase table.

Figure 6(top-left) demonstrates the request latencies of the three high priority workflows, normalized to their SLOs. During each of the three phases of the experiment, a background workflow temporarily increases its request

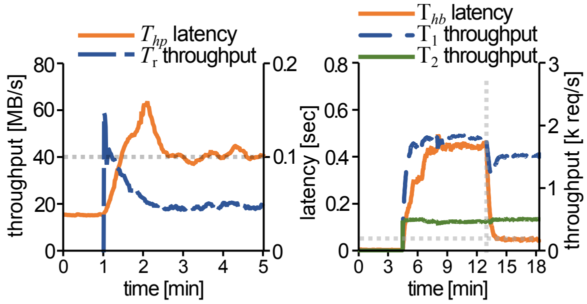


Figure 7: LATENCYSLO rate-limits replication to enforce a 100ms SLO for T_{hp} (left). LATENCYSLO enforces a 50ms latency for heartbeats (right).

rate, affecting the latency of the high priority workflows. In the first stage, F_4 increases its load and F_1 and F_2 miss their SLO. In the second stage, F_5 increases its load and F_1 misses its SLO by a factor of 10. In the last stage, F_6 increases its load and F_2 and F_3 miss their SLOs by factors of 10 and 500 respectively. Figure 6(bottom-left), shows the slowdown of different resources as the experiment progresses: at first F_4 table scans cause disk slowdown, then F_5 causes HDFS NameNode lock and NameNode queue slowdown, and finally F_6 causes CPU and HBase queue slowdown as its data is cached.

We repeat the experiment using LATENCYSLO to enforce the SLOs of F_1, F_2 and F_3. Figure 6(top-right) shows that the policy successfully maintains the SLOs by throttling the background workflows at a number of control points within HDFS and HBase. Figure 6(bottom-right) shows the sparklines of the workflow *utilization ratios* – the achieved throughput relative to the allocated rate at a particular control point, similar to Figure 5. We see that LATENCYSLO only rate-limits the background workflows during their specific overload phases.

These results highlight how LATENCYSLO selectively throttles workloads based on their contribution to the SLO violation. Retro can enforce SLOs for multiple workflows across software and hardware resources simultaneously.

Background workflows Thanks to the workflow abstraction, LATENCYSLO is equally applicable to providing guarantees for high priority background tasks, such as heartbeats, or to protecting high priority workflows from aggressive background tasks such as data replication.

Figure 7(right) demonstrates the effect of two workflows T_1 and T_2 on the latency of datanode heartbeats, T_{hb}. The heartbeat latency increases from 4ms to about 450ms when T_1 and T_2 start renaming files and listing directories, respectively, causing increased load the HDFS namesystem lock. Whilst T_{hb} and T_2 only require read locks, T_1 requires write locks to update the filesystem, thus blocking heartbeats. When we start SLO enforcement at t=13, the policy identifies T_1 as the cause of slowdown, throttles it at the NameNode RPC queue, and achieves the heartbeat SLO.

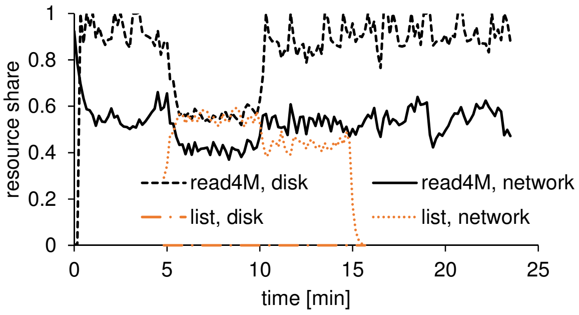


Figure 8: Resource share for experiment described in §6.3.

In Figure 7(left), LATENCYSLO rate-limits low-priority background replication T_r, to provide guaranteed latency to high priority workflow T_{hp} submitting 8kB read requests with 100ms SLO. At t=1, we manually trigger replication of a large number of HDFS blocks; subsequently, LATENCYSLO rate-limits T_r. High-priority replication (single remaining replica) could use a separate workflow ID to avoid throttling.

6.3 RDRF in HDFS

To demonstrate RDRF (Figure 8), we run an experiment with two workflows – READ4M with 50 clients reading 4MB files, and LIST with 5 clients listing 1000 files in a directory – accessing the HDFS cluster remotely sharing a 1Gbps network link. The dominant resource for READ4M is disk and for LIST it is the network, since it is reading large amounts of data from the memory of the NameNode.

We start READ4M at t=0 and add LIST at t=5, with sharing weights of 1. Between time 5 and 10, RDRF throttles READ4M to achieve equal dominant shares across both of these workflows (60% on disk and network). After increasing the weight of READ4M to 2 at t=10, the dominant shares change to 80% and 40%, respectively.

Despite knowing neither the demands of each workflow, nor the capacity of each resource, RDRF successfully allocates each workflow the fair share of its dominant resource. The experiment demonstrates how slowdown is viable as a proxy for resource capacity, and coupled with reactive policies, enables us to overcome some limitations of an existing resource fairness technique.

6.4 Overhead and scalability of Retro

Retro propagates a workflow ID (3 bytes) along the execution path of a request, incurring up to 80ns of overhead (see Table 2) to serialize and deserialize when making network calls. The overhead to record a single resource operation is approximately 340ns, which includes intercepting the thread, recording timing, CPU cycle count (before and after the operation), and operation latency, and aggregating these into a per-workflow report.

To estimate the impact of Retro on throughput and end-to-end latency, we benchmark HDFS and HDFS instrumented with Retro using requests derived from the

Operation	Latency
Deserialize metadata	80ns
Read active metadata	9ns
Serialize metadata	46ns
Record use one resource operation	342ns

Table 2: Costs of Retro instrumentation

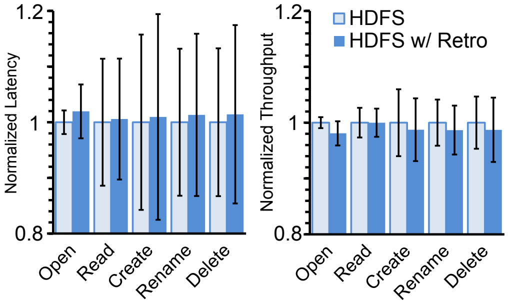


Figure 9: Normalized latency (left) and throughput (right) for HDFS NameNode benchmark operations along with error bars showing one standard deviation.

HDFS NNBench benchmark. See Figure 9 for throughput and end-to-end latency for five requests types. *Open* opens a file for reading; *Read* reads 8kB of data from a file; *Create* creates a file for writing; *Rename* renames an existing file and *Delete* deletes the file from the name system (and triggers an asynchronous block delete). Of the request types, *Read* is a DataNode operation and the others are NameNode operations. In all cases, latency increases by approximately 1-2%, and throughput drops by a similar 1-2%. Variance in latency and throughput increases slightly in HDFS instrumented with Retro. These overheads could be further significantly reduced by *sampling*, *i.e.*, tracing only a subset of requests or operations.

We evaluate Retro's ability to scale beyond the cluster sizes presented thus far with an 200-VM experiment on Windows Azure (Standard_A2 instances). Figure 10 shows slowdown and aggregate disk throughput for four workflows when BFAIR is activated (at t=1.5) and per-workflow weights are adjusted (at t=4). Each workflow ran a mix of 64MB HDFS reads and writes, with 800, 1200, 1600, and 2000 closed-loop clients respectively. Before the policy is activated we observe the expected imbalance in disk throughput caused by the differing number of clients in each workflow. When the policy is activated at t=1.5, the workflows quickly converge to an equal share of disk throughput, and the slowdown decreases to the target of 50. At t=4, two of the clients are given a weight of 2 and the policy quickly establishes the new fair share.

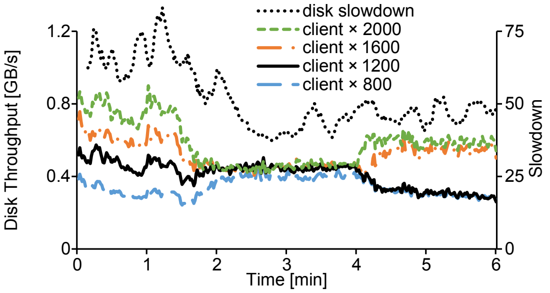


Figure 10: Retro's BFAIR policy on a 200-node cluster with four workflows and overloaded disks. BFAIR is enabled at t=1.5 with a target slowdown of 50; client weights are adjusted at t=4.

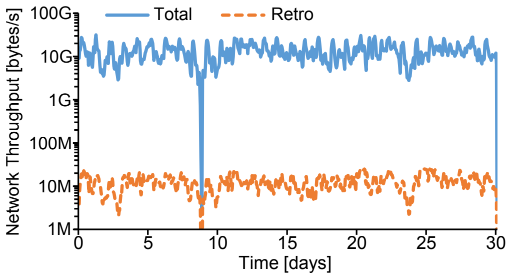


Figure 11: Total network throughput for a several-hundred node production Hadoop cluster and network throughput of Retro, calculated from 1 month of traces. Retro's bandwidth requirements are on average 0.1% of the total throughput.

We evaluate the scalability of Retro's central controller in terms of its ability to process resource reports. In a benchmark where each report contains resource usage for 1000 workflows, the controller can process on the order of 10,000 reports per second. Assuming 10 resources per machine, the controller could thus support up to 1000 machines. In this setup, each machine would use about 600kB/sec of network bandwidth to send the reports. Figure 11 shows calculated network overhead that would be imposed by Retro on a production Hadoop cluster comprising several hundred nodes over a period of a month. We calculate the network traffic that would be generated by Retro based on traces from this production cluster. The figure shows that Retro would account for an average of 0.1% of the network traffic present. Furthermore, since Retro aggregation only computes sums and averages, we can aggregate hierarchically (*e.g.* inside each machine and rack), further reducing the required network bandwidth and thereby supporting much larger deployments.

Whilst Retro requires manual developer intervention to propagate workflow IDs across network boundaries and to verify correct behavior of Retro's automatic instrumentation, our experience shows that this requires little work. For example, instrumenting each of the five systems required only between 50 and 200 lines of code; for example to handle RPC messages. Instrumenting resource operations happens automatically through AspectJ.

7 Discussion

In Retro, we made the decision to implement both resource measurement and control points at the application level. While applying Retro in the OS, hypervisor, or device driver level could provide more accurate measurements and fine-granularity enforcement, our approach has the advantages of fast and pervasive deployment, and of not requiring specially built OS or drivers (we deployed Retro in both Windows and Linux environments). Retro's promising results indicate that OS's, and distributed systems in general, should provide mechanisms to facilitate the propagation of workflow IDs across their components.

Retro is extensible to handle *custom resources*. For example, in systems with row-level locking we cannot treat each lock/row as an individual resource because the number of resources might be unbounded. Instead, we could define each data partition as a logical resource, which would significantly reduce the number of resources in the system. ZooKeeper uses a custom request processing pipeline, which is not part of Java standard library. We treat ZooKeeper queues as custom resources and estimate their load and slowdown.

The current implementation of Retro has several limitations. First, some resources cannot be automatically *revoked* once a request has obtained them and have to be explicitly released by the system. For example, this applies to memory, sockets, or disk space. A developer could implement application-specific hooks that Retro could use to reclaim resources. Second, because the rates of distributed token buckets are updated only once a second, when workload is very variable, this might reduce the throughput of the system. Using different local schedulers, such as weighted fair queues [35] and reservations [15] would alleviate this problem.

8 Related work

In [26] we introduced the design principles behind Retro, as well as a preliminary implementation of resource measurement for HDFS. This paper presents a complete framework by adding the centralized controller, resource management policies, and distributed control points, evaluated across five different distributed systems.

Multi-resource scheduling Several research projects tackle multi-resource allocation, such as Cake [44], mClock [15], IOFlow [39], and SQLVM [27]. These frameworks are specific to particular systems, such as storage or relational databases. Retro improves on top of these by providing the workflow, resource, and control point abstractions, which allow it to handle a wide range of resources and system activities, and enforce policy decisions across the whole system.

Cake provides isolation between low-latency and high-throughput tenants using HBase and HDFS. However, it treats HDFS as a single resource, and cannot target specific resource bottlenecks and workflows that overload these resources. mClock is a disk IO scheduler that could be implemented as a Retro control point. IOFlow provides per-tenant guarantees for remote disk IO requests in datacenters but does not schedule other resources such as threadpools, CPU, and locks. SQLVM [27] provides isolation for CPU, disk IO, and memory for multiple relational databases deployed in a single machine, but does not deal with distributed scenarios.

In the data analytics domain, task schedulers such as Mesos [20], Yarn [42], or Sparrow [29] use a centralized approach to allocate individual tasks to machines. In these frameworks, each task passes through the scheduler before starting its execution, the scheduler can place it to an arbitrary machine in the cluster and after starting execution, the task is not scheduled any more. In typical distributed systems, requests do not pass through a single point of execution and routing of a request through the system is driven by complex internal logic. Finally, to achieve fine-grained control over resource consumption, requests have to be throttled *during* its execution, not only at the beginning. These frameworks thus do not directly apply to scheduling in general distributed systems. On the other hand, Retro requires no knowledge of internal design of the system and provides fine-grained throttling using control points on the request execution path.

End-to-end (resource) tracing Banga and Druschel addressed the mismatch between OS abstractions and the needs of resource accounting with resource containers [4], which, albeit in a single machine, aggregate resource usage orthogonally to processes, threads, or users. Our end-to-end propagation of workflow IDs shares mechanisms with taint tracking [28] and several causal tracing frameworks [5, 8, 9, 11, 34, 37, 40]. Retro does not, however, record causality or traces, but rather uses the workflow information to attribute resource usage. Whodunit [7] uses causal propagation to record timings between parts of a program, and provides a profile of where requests spent their time. Timecard [33] also propagates cumulative time information in the request path between a mobile web client and a server, and uses this in real time to speed up the processing of requests that are late. Retro, in contrast, records aggregate resource profiles by workflow and uses these to enforce flexible high-level policies.

9 Conclusion

Retro is a framework for implementing resource management policies in multi-tenant distributed systems. Retro tackles important challenges and provides key abstractions that enable a separation between resource-management policies and mechanisms. It requires low developer effort, and is lightweight enough to be run in production. We demonstrate the applicability of Retro to key components of the Hadoop stack and develop and evaluate three targeted and reactive policies for achieving fairness and latency targets. These policies are system-agnostic, resource-agnostic, and uniformly treat all system activities, including background management tasks. To the best of our knowledge, Retro is the first framework to do so. We plan to extend the control points to provide fair scheduling, prioritization, and load balancing.

References

[1] Intel Performance Counter Monitor – A better way to measure CPU utilization. http://intel.ly/1C23e67.

[2] F. Akgul. *ZeroMQ*. Packt Publishing, 2013.

[3] Amazon web services. http://aws.amazon.com/.

[4] G. Banga, P. Druschel, and J. C. Mogul. Resource containers: a new facility for resource management in server systems. In *OSDI '99*, pages 45–58, Berkeley, CA, USA, 1999. USENIX Association.

[5] P. Barham, A. Donnelly, R. Isaacs, and R. Mortier. Using Magpie for Request Extraction and Workload Modeling. In *Proc. USENIX OSDI*, 2004.

[6] B. Calder, J. Wang, A. Ogus, N. Nilakantan, A. Skjolsvold, S. McKelvie, Y. Xu, S. Srivastav, J. Wu, H. Simitci, et al. Windows Azure Storage: a highly available cloud storage service with strong consistency. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, pages 143–157. ACM, 2011.

[7] A. Chanda, A. L. Cox, and W. Zwaenepoel. Whodunit: Transactional Profiling for Multi-Tier Applications. In *EuroSys'07*, Lisbon, Portugal, March 2007.

[8] A. Chanda, K. Elmeleegy, A. L. Cox, and W. Zwaenepoel. Causeway: System support for controlling and analyzing the execution of multi-tier applications. In *Proc. Middleware 2005*, pages 42–59, November 2005.

[9] M. Chen, E. Kiciman, E. Fratkin, E. Brewer, and A. Fox. Pinpoint: Problem Determination in Large, Dynamic, Internet Services. In *Proc. International Conference on Dependable Systems and Networks*, 2002.

[10] T. Do, H. S. Gunawi, T. Do, T. Harter, Y. Liu, H. S. Gunawi, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. The case for limping-hardware tolerant clouds. In *5th USENIX Workshop on Hot Topics in Cloud Computing (HotCloud)*, 2013.

[11] R. Fonseca, G. Porter, R. H. Katz, S. Shenker, and I. Stoica. X-trace: A pervasive network tracing framework. In *Proceedings of the 4th USENIX Conference on Networked Systems Design &#38; Implementation*, NSDI'07, Berkeley, CA, USA, 2007. USENIX Association.

[12] A. Ghodsi, V. Sekar, M. Zaharia, and I. Stoica. Multi-resource fair queueing for packet processing. In *Proceedings of the ACM SIGCOMM 2012 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication*, SIGCOMM '12, pages 1–12, New York, NY, USA, 2012. ACM.

[13] A. Ghodsi, M. Zaharia, B. Hindman, A. Konwinski, S. Shenker, and I. Stoica. Dominant resource fairness: fair allocation of multiple resource types. In *USENIX NSDI*, 2011.

[14] Google Protocol Buffers. http://code.google.com/p/protobuf/.

[15] A. Gulati, A. Merchant, and P. J. Varman. mClock: Handling Throughput Variability for Hypervisor IO Scheduling. In R. H. Arpaci-Dusseau and B. Chen, editors, *Proceedings of OSDI*, pages 437–450. USENIX Association, 2010.

[16] Z. Guo, S. McDirmid, M. Yang, L. Zhuang, P. Zhang, Y. Luo, T. Bergan, M. Musuvathi, Z. Zhang, and L. Zhou. Failure recovery: When the cure is worse than the disease. In *Presented as part of the 14th Workshop on Hot Topics in Operating Systems*, Berkeley, CA, 2013. USENIX.

[17] HBase. http://hbase.apache.org.

[18] HDFS-4183. http://bit.ly/1l4uWbu.

[19] HDFS API. http://bit.ly/1cxFTD9.

[20] B. Hindman, A. Konwinski, M. Zaharia, A. Ghodsi, A. D. Joseph, R. Katz, S. Shenker, and I. Stoica. Mesos: A platform for fine-grained resource sharing in the data center. In *Proceedings of the 8th USENIX Conference on Networked Systems Design and Implementation*, NSDI'11, Berkeley, CA, USA, 2011. USENIX Association.

[21] P. Hunt, M. Konar, F. P. Junqueira, and B. Reed. Zookeeper: wait-free coordination for internet-scale systems. In *Proceedings of the 2010 USENIX conference on USENIX annual technical conference*, volume 8, 2010.

[22] T. Johnson. Approximate analysis of reader/writer queues. *IEEE Trans. Softw. Eng.*, 21(3):209–218, Mar. 1995.

[23] H. Kang. *Computational Color Technology*. Press Monographs. Society of Photo Optical, 2006.

[24] S.-I. Kang and H.-K. Lee. Analysis and solution of non-preemptive policies for scheduling readers and writers. *SIGOPS Oper. Syst. Rev.*, 32(3):30–50, July 1998.

[25] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. G. Griswold. An Overview of AspectJ. In *Proceedings of the 15th European Conference on Object-Oriented Programming*, ECOOP '01, pages 327–353, London, UK, UK, 2001. Springer-Verlag.

[26] J. Mace, P. Bodik, R. Fonseca, and M. Musuvathi. Towards general-purpose resource management in shared cloud services. In *10th Workshop on Hot Topics in System Dependability (HotDep 14)*, Broomfield, CO, Oct. 2014. USENIX Association.

[27] V. R. Narasayya, S. Das, M. Syamala, B. Chandramouli, and S. Chaudhuri. Sqlvm: Performance isolation in multi-tenant relational database-as-a-service. In *CIDR'13*. www.cidrdb.org, 2013.

[28] J. Newsome and D. Song. Dynamic taint analysis for automatic detection, analysis, and signature generation of exploits on commodity software. In *Proceedings of the 12th Annual Network and Distributed System Security Symposium (NDSS '05)*, Feb. 2005.

[29] K. Ousterhout, P. Wendell, M. Zaharia, and I. Stoica. Sparrow: Distributed, low latency scheduling. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, SOSP '13, pages 69–84, New York, NY, USA, 2013. ACM.

[30] J. Padhye, V. Firoiu, D. Towsley, and J. Kurose. Modeling TCP throughput: A simple model and its empirical validation. In *ACM SIGCOMM Computer Communication Review*, volume 28(4), pages 303–314. ACM, 1998.

[31] A. K. Parekh and R. G. Gallagher. A generalized processor sharing approach to flow control in integrated services networks: the multiple node case. *IEEE/ACM Transactions on Networking (TON)*, 2(2):137–150, 1994.

[32] L. C. Puryear and V. G. Kulkarni. Comparison of stability and queueing times for reader-writer queues. *Perform. Eval.*, 30(4):195–215, 1997.

[33] L. Ravindranath, J. Padhye, R. Mahajan, and H. Balakrishnan. Timecard: Controlling user-perceived delays in server-based mobile applications. In *SOSP '13*, pages 85–100, New York, NY, USA, 2013. ACM.

[34] 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'06*, Berkeley, CA, USA, 2006. USENIX Association.

[35] M. Shreedhar and G. Varghese. Efficient fair queuing using deficit round-robin. *Networking, IEEE/ACM Transactions on*, 4(3):375–385, 1996.

[36] K. Shvachko, H. Kuang, S. Radia, and R. Chansler. The Hadoop distributed file system. In *Mass Storage Systems and Technologies (MSST), 2010 IEEE 26th Symposium on*, pages 1–10. IEEE, 2010.

[37] B. H. Sigelman, L. A. Barroso, M. Burrows, P. Stephenson, M. Plakal, D. Beaver, S. Jaspan, and C. Shanbhag. Dapper, a large-scale distributed systems tracing infrastructure. Technical report, Google, Inc., 2010.

[38] D. Stiliadis and A. Varma. Latency-rate servers: a general model for analysis of traffic scheduling algorithms. *IEEE/ACM Transactions on Networking (ToN)*, 6(5):611–624, 1998.

[39] E. Thereska, H. Ballani, G. O'Shea, T. Karagiannis, A. Rowstron, T. Talpey, R. Black, and T. Zhu. IOFlow: A Software-defined Storage Architecture. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, SOSP '13, pages 182–196. ACM, 2013.

[40] E. Thereska, B. Salmon, J. Strunk, M. Wachs, M. Abd-El-Malek, J. Lopez, and G. R. Ganger. Stardust: Tracking activity in a distributed storage system. *SIGMETRICS Perform. Eval. Rev.*, 34(1):3–14, June 2006.

[41] A. Thusoo, J. Sarma, N. Jain, Z. Shao, P. Chakka, N. Zhang, S. Antony, H. Liu, and R. Murthy. Hive – a petabyte scale data warehouse using Hadoop. In *ICDE'10*, pages 996 –1005, march 2010.

[42] V. K. Vavilapalli, A. C. Murthy, C. Douglas, S. Agarwal, M. Konar, R. Evans, T. Graves, J. Lowe, H. Shah, S. Seth, B. Saha, C. Curino, O. O'Malley, S. Radia, B. Reed, and E. Baldeschwieler. Apache Hadoop YARN: Yet Another Resource Negotiator. In *Proceedings of the 4th Annual Symposium on Cloud Computing*, SOCC '13, pages 5:1–5:16, New York, NY, USA, 2013. ACM.

[43] A. Wang. Personal communication.

[44] A. Wang, S. Venkataraman, S. Alspaugh, R. Katz, and I. Stoica. Cake: Enabling High-level SLOs on Shared Storage Systems. In *Proc. SoCC*. ACM, 2012.

Scalable error isolation for distributed systems

Diogo Behrens,[*] Marco Serafini,[◇] Sergei Arnautov,[*] Flavio P. Junqueira,[‡] Christof Fetzer[*]

[*]*Technische Universität Dresden, Germany*

[◇]*Qatar Computing Research Institute, Qatar*

[‡]*Microsoft Research, Cambridge, UK*

Abstract

In distributed systems, data corruption on a single node can propagate to other nodes in the system and cause severe outages. The probability of data corruption is already non-negligible today in large computer populations (*e.g.*, in large datacenters). The resilience of processors is expected to decline in the near future, making it necessary to devise cost-effective software approaches to deal with data corruption.

In this paper, we present SEI, an algorithm that tolerates Arbitrary State Corruption (ASC) faults and prevents data corruption from propagating across a distributed system. SEI scales in three dimensions: memory, number of processing threads, and development effort. To evaluate development effort, fault coverage, and performance with our library, we hardened two real-world applications: a DNS resolver and `memcached`. Hardening these applications required minimal changes to the existing code base, and the performance overhead is negligible in the case of applications that are not CPU-intensive, such as `memcached`. The memory overhead is negligible independent of the application when using ECC memory. Finally, SEI covers faults effectively: it detected all hardware-injected errors and reduced undetected errors from 44% down to only 0.15% of the software-injected computation errors in our experiments.

1 Introduction

Distributed systems running in modern data centers must be tolerant to faults. Since machine and process crashes are commonly observed, the crash fault model is the one typically adopted. In fact, many systems critical to Web-scale online services are successfully using techniques such as state machine replication [6, 17, 22, 31] to guarantee availability despite crash faults.

The crash model does not cover data corruption faults, which might lead to errors propagating through incorrect messages after a fault. Incidents occurring in large Internet services in the recent past already indicate that data corruption can cause process state corruption [20], data loss [2, 3, 4], or in some unlucky case of error propagation, even multi-hour outages of entire services [1]. This is not surprising: several large scale studies show that faults that would be very unlikely in a small cluster become much more likely at scale and tend to reappear more frequently after the first occurrence [25, 32, 43, 44, 50]. There can be several reasons for data corruption symptoms, for example, manufacturing problems, overheating, an incorrect use of dynamic voltage scaling, hardware/software incompatibility, or power supply faults [2, 36]. For example, we used dynamic voltage scaling while running `memcached` on a single processor with a lower voltage level and found that undetected error propagation occurred in 4 out of 468 runs (see Section 7.2). These problems are in fact known to datacenter operators dealing with large server populations.

New processor generations have traditionally achieved higher performance through higher circuit density and lower energy consumption. This approach, however, has reached physical limits that affect hardware-level reliability negatively [15, 21]. The rate of transient errors for processors has been rising [14, 16] and it might reach up to one user-visible failure per day per chip with 16 nm technology [26, 51].

These trends are already changing the way large-scale distributed systems are designed today. Mesa, a data warehousing system for business-critical data used in Google, uses application-level integrity checks to deal with transient data corruption during computation, which is common at scale [28].

We argue that preventing end-to-end error propagation due to data corruption, including corruption in the computation, is an important requirement for large-scale fault-tolerant distributed systems. Application-specific solutions like the ones used in Mesa leave application developers with the burden of guaranteeing data integrity.

Instead, we advocate for *hardening* approaches that enable crash-tolerant systems to handle data corruption with minimal changes to the application code base.

Existing hardening solutions are not sufficiently cost-effective at scale: they either rely on expensive servers with hardware-level redundancy [7, 53] or require process replication and coordination over multiple physical machines even to achieve simple error isolation [30]. Recent software-based approaches ensure end-to-end error isolation in distributed systems without physical replication, but they still increase the application state by a factor of two and do not support multithreading [12, 23]. The widespread use of multi-core machines with large main memory requires solutions that scale well with many application threads and a large in-memory state.

We introduce Scalable Error Isolation (SEI), a scalable hardening algorithm that transforms the processes of an arbitrary event-based distributed system to ensure *error isolation*. With error isolation, local errors cannot propagate to other processes via output messages in an undetectable manner. SEI is designed to formally guarantee error isolation in the presence of Arbitrary State Corruption (ASC) faults, a well defined and general fault model [23]. We have implemented a C library called `libsei` to harden distributed systems using the SEI algorithm. Hardening is semi-automated: a developer simply annotates, using the `libsei` API, the portions of the code of a distributed system process that are responsible for handling messages, as well as the input and output messages. The compiler then automatically hardens the implementation by instrumenting it with `libsei`.

SEI and `libsei` are scalable in three dimensions. For **memory**, the additional state and redundant information they use is small and independent of the memory usage of the hardened process. For **computation**, they support multithreading and thus cover complex error propagation patterns among threads sharing the same state. For **development effort**, they enable hardening real-world applications with minor developer involvement.

SEI detects *data corruption in the computation*, *i.e.*, errors in the arithmetic and logic units of the processors and in the register files, by processing each message twice and comparing the results locally. The SEI hardening code itself is untrusted, so checks might be incorrect or skipped due to faults. Consequently, SEI runs multiple integrity checks for data and control-flow errors.

SEI detects *data corruption in memory* using compact error detection codes. These codes can be implemented in software, but SEI can also leverage hardware-level mechanisms, such as ECC or Chipkill in DRAM memory modules, to virtually eliminate the memory overhead. These hardware mechanisms both perform this part of the hardening very efficiently and are effective for data in memory [32, 50]. Given that there is no expectation that memory error rates will increase [15, 51], they are likely to remain effective.

To show that hardening existing systems with a small amount of effort is possible, we have hardened real-world applications: `memcached`, a popular in-memory distributed cache system, and Deadwood, a recursive DNS resolver. Hardening these systems required a good understanding of the code base but only small changes.

We conducted extensive fault injection experiments, both software- and hardware-implemented, and a performance evaluation of our hardened applications. We injected faults at hardware level by reducing the CPU voltage and observed that SEI detected all errors. We also injected targeted data corruption faults *during computation* and found that SEI makes the likelihood of error propagation under these faults negligible: from 44% down to only 0.15% of the errors.

Performance results show that the overhead depends on the original bottlenecks of the system. In the case of `memcached`, the application is not CPU-bound so there are spare cycles available for additional processing and the overhead is negligible. Deadwood, however, is CPU intensive so its throughput is reduced to nearly one half.

The remainder of this paper is structured as follows. Section 2 discusses related work. Section 3 introduces the system and fault models. Section 4 presents the SEI algorithm. Sections 5 and 6 discuss the `libsei` implementation and our experience hardening `memcached` and Deadwood. Sections 7 and 8 present our fault injection and performance results. Section 9 concludes this work.

2 Related work

We now discuss the most related approaches for error isolation in distributed systems.

Byzantine fault tolerance. Given the body of work on Byzantine Fault Tolerance (BFT), it is natural to consider Byzantine faults to cover data corruption [18, 37, 40]. The Byzantine model assumes a powerful adversary, and consequently a Byzantine-tolerant system is able to cope with data corruption. Byzantine-tolerant protocols, however, also tolerate faults that are orthogonal to resilience against data corruption, such as intrusions or bugs, as long as replicas use diverse implementations to guarantee fault independence [19]. In particular, intrusions fall into the domain of security, which is often treated as a separate concern in data center applications [13].

Deadwood and `memcached` are instances of a large class of systems in which integrity (safety) is sufficient and continuous availability is not strictly necessary. In the Byzantine model, providing just safety does not significantly reduce cost; see for example Nysiad, which achieves safety through replication and agreement [30].

The Thema system also shows the additional complexity of building Byzantine-tolerant three-tiered Web systems [42]. In contrast to BFT, SEI is lightweight, not requiring replication nor complex agreement protocols. SEI can still be used to harden replicated systems, *e.g.*, replicated state machines based on Paxos [39]. Finally, BFT primarily targets single-threaded state machines. Eve runs multithreaded state machines in multi-core systems by leveraging commutative operations [35]. SEI supports regular multithreaded applications and does not rely on commutativity.

Software-level error detection. Software-level error detection has been subject of a large body of research work. Most such techniques do not provide end-to-end guarantees in distributed systems under ASC faults (see for example SWIFT [48] or [47]). Recent work has proposed using encoded execution to harden distributed systems and provide end-to-end guarantees [12]. Encoding presents important drawbacks, however. First, it induces a significant overhead, showing a response time increased up to 20 times even at modest request loads. For a service such as `memcached` that is sensitive to latency, such an overhead is not acceptable. Second, encoding blows up the application state by a factor of two. Optimizations for both issues exist [10], but are limited to state machine replication. In contrast, SEI presents moderate overhead and has a small memory footprint with hardware error detection codes.

PASC is a hardening algorithm that, like SEI, guarantees coverage of ASC faults, is untrusted, and does not require physical replication [23]. PASC can be used to harden state machine replication: a comparison between BFT and an ASC-tolerant version of Paxos is presented in Correia *et al.* [23]. However, PASC is not scalable in any of the three dimensions indicated by SEI: it does not support processes with multiple threads, it doubles the memory requirements of the hardened application, and it requires implementing the distributed system from scratch using a fixed template. The first two points limit its applicability to multithreaded, memory-intensive systems like `memcached`. The last point makes it hard to harden existing code bases, also because all state accesses must be mediated by a single state object [9].

3 System and fault model

This section presents an informal description of our system and fault model, which is an adaptation of the ASC fault model [23] to multithreaded settings. We refer to our technical report [11] for a complete formalization.

System model. SEI targets event-based processes of distributed systems. Processes consist of one or more threads that spin over three phases:

- *Dispatching* receives a new event (message) and selects an event handler;
- *Handling* executes the actual system logic;
- *Output* sends out messages produced by the event handler.

Threads read from and write to *state variables*, which collectively form the state of the process. These variables persist across the multiple event handling cycles and can be shared among threads. A thread might also have a *local state*, which encompasses the variables that are instantiated every time a handler is executed, but do not persist across handler executions. The state of a process includes all state that is directly observed by its threads and used to determine their behavior. In this work, we consider only state stored in memory, but the model could also be extended to disk storage.

The event handling logic is deterministic, *i.e.*, the state updates and outputs it produces depend uniquely on the input message and the values returned by its reads from the process state. However, we do not require deterministic thread scheduling. Threads can be scheduled in any order and preempted arbitrarily.

Threads can interact through shared variables, which are only accessed in critical sections protected by locks. While this requirement does not cover applications using lock-free state sharing, it represents a very common approach. We assume that threads use lock hierarchies [29], a standard technique to avoid circular waits and deadlocks. Lock hierarchies determine a fixed total order among all locks, and threads acquire and release the locks they need according to this order.

Fault model. We consider a conservative fault model for transient hardware faults. An *Arbitrary State Corruption (ASC) fault* can either crash the process or modify its state by assigning an arbitrary value to any number of its variables. A fault can also corrupt the program counter and make it jump to a different instruction.

This fault model admits worst-case state corruption scenarios. Faults can modify any number of variables and assign them any value. Corruption can occur while data is stored, for example, in main memory, or data is computed, for example, by the combinational logic of a processor. Since it is difficult to determine precisely which part of a process state can be corrupted by a hardware malfunction, a worst-case model is easier to generalize over different applications and platforms.

To guarantee data integrity, the ASC model assumes that it is possible to implement reliable *integrity checks* to detect data corruption while data is stored, for example, in memory, or transmitted as a message. Formally, a variable v in the process state is accessed with the instructions $read(v)$ and $write(v, val)$, where val is the new value of v. The integrity check of a variable v is performed by calling *check(v)*. The model assumes the following *corruption coverage* property of integrity checks: if the current value of a variable v has been determined by an ASC fault, then *check(v)* detects the error by evaluating to FALSE. The ASC model does not specify how integrity checks are implemented. Possible options are Cyclic Redundant Codes (CRC) or hardware techniques like ECC memory. The corruption coverage property is consistent with these techniques: *check(v)* cannot detect errors if the current value *val* of v has been written by a $write(v, val)$ instruction. With an ECC implementation of integrity checks, errors that are detected and not corrected correspond to *check(v)* calls returning FALSE right before *read(v)* calls. The $write(v, val)$ calls overwrite the memory locations storing v and update their ECCs.

ASC faults cannot occur infinitely often. An execution of the system comprises an unbounded number of thread steps. Such an execution might include an unbounded number of faults, but at most one fault occurs during any event-handling phase of any thread (multiple faults can still occur before that thread starts the next event-handling phase). This assumption does not limit the number of faults in an execution, but it limits *fault frequency*; we justify this assumption as follows. The distributed systems we target handle events such as processing an incoming message in no more than a few milliseconds. The fault model consequently assumes that no two faults happen within such a short time window. The frequency of *uncorrectable* hardware-level data corruption previously reported indicates that this assumption holds with very high probability [32, 43, 50]. It holds even if we consider *hard* errors that occur intermittently at the same memory location [32].

After an ASC fault, computation is expected to continue according to the specification, although perhaps from an instruction that is inconsistent with the previous execution flow and from a wrong state. Transient text-segment corruption typically results in ASC faults: an incorrect operation might corrupt some of the operands, update the wrong variable, or result in an incorrect jump. Related work on the Ensemble system confirms this observation [8]. These faults are often enough tolerated by ASC-hardening, and in fact, the injection of text-segment faults on an ASC-hardened Paxos implementation has shown no case of error propagation [23]. In our hardened `memcached`, text-segment corruption resulted in error propagation with negligible probability (3 out of 7000 errors). There are, however, some cases of text-segment corruption, like some corruptions of load and store instructions, that cannot be tolerated in our model.

4 The SEI hardening algorithm

The SEI algorithm takes an event-based process as specified in the previous section and transforms it into a *hardened process*. A hardened process executes the same application logic as the original process, together with additional checks against ASC faults.

The SEI algorithm comprises a set of transformation rules that introduce both redundant execution and additional verification steps to the original code. To prevent error propagation, SEI might induce a faulty process to abort (*i.e.*, crash) if an internal corruption is detected, or might make a correct process discard a corrupt message it has received. While a faulty process might still send out incorrect messages, incorrect messages do not appear correct to receivers.

In a distributed system where all processes are hardened, SEI guarantees the following properties [11]:

- *Error isolation:* A correct process discards any corrupt input message.
- *Accuracy:* Hardening never causes a correct process to crash or discard a correct input message.

Messages are either correct or corrupt. A correct message is informally defined as follows. Let p be a process and s be the sequence of correct messages p received before sending a message m. Message m is *correct* if and only if there exists a subsequence h_m of s, called a *generation history*, such that the correct behavior of p after receiving only h_m would be to output m. By induction, each generation history of each output message that p produces after sending m extends some generation history of m. In presence of multiple threads processing input messages in parallel, there exists also a consistent interleaving of steps that generates all output messages.

With error isolation, a distributed system designed to tolerate crashes and message omissions is guaranteed to also tolerate ASC faults once its processes are hardened. Accuracy rules out trivial ways to achieve error isolation.

Before discussing details of SEI, we show with an example how to harden an event handler against a small subset of ASC faults by progressively adding checks.

4.1 SEI by example

We illustrate the hardening transformation by presenting an example in which a simple event handler eliminates all but the least significant digit (modulo 10) of the state variable X using a temporary variable V (see Figure 1a). In this example, a fault might corrupt X before the process sends out the message containing X and propagates the incorrect value of X to another process.

A first improvement is to duplicate instructions and variables (see Figure 1b). This approach resembles EDDI [46] and SWIFT [48]. Instruction duplication prevents error propagation if a fault only corrupts *one* variable (*i.e.*, V, V', X, or X') *before* the check of Line 6. Instruction duplication, however, cannot detect several other corruption scenarios. As an example, consider

a) Original code

```
   // handler
1  V ← X
2  V ← V % 10
3  X ← V
   // send message
4  send X
```

b) Instruction duplication

```
   // handler
1  if X ≠ X’ then Abort
2  V ← X
3  V’ ← X’
4  V ← V % 10
5  V’ ← V’% 10
6  if V ≠ V’ then Abort
7  X ← V
8  X’ ← V’
   // send message
9  send X
```

c) PASC-like

```
   // Execution 1
1  if X ≠ X’ then Abort
2  V ← X
3  V ← V % 10
4  N ← V
   // Execution 2
5  if X ≠ X’ then Abort
6  V ← X’
7  V ← V % 10
8  X’ ← V
   // Validation
9  if N ≠ X’ then Abort
   // Final update
10 X ← N
   // send message
11 send X, CRC(X’)
```

d) Simplified SEI

```
   // Execution 1
1  V ← X if X = X’ else Abort
2  V ← V % 10
3  O ← X if X = X’ else Abort
4  X, X’ ← V
   // Reset
5  N ← V
6  X, X’ ← O
   // Execution 2
7  V ← X if X = X’ else Abort
8  V ← V % 10
9  X, X’ ← V
10 C ← CRC(V)
   // Validation
11 if N ≠ X then Abort
   // send message
12 send X, C
```

Figure 1: Hardening of a handler that updates a state variable X via a temporary variable V, then sends X out. Primed variables are replica variables (*e.g.*, X' is a replica of X). Comparisons among replicas (*e.g.*, **if** $X = X'$) implement the $check(v)$ operation. "←" represents either $read(v)$ or $write(v, val)$ or a combination of them.

a *last-mile fault*: a fault corrupting V or X between Lines 6 and 9. Even adding a further check comparing X to X' before Line 9 does not help because a fault could still occur after the check.

Another example where instruction duplication falls short is *multi-variable corruption* (recall that a single ASC fault may corrupt any number of variables). By the corruption coverage property, an ASC fault alone cannot corrupt X and make $check(X)$ true, *i.e.*, a fault corrupting X and X' has to result in $X \neq X'$. However, if an ASC fault corrupts X and X' between Lines 1 and 2, then the execution of the subsequent instructions can let V and V' have incorrect but equal values in Line 6. For example, if initially $X = X' = 101$ and an ASC fault between Lines 1 and 2 results in $X = 10$ and $X' = 20$, then $V = V' = 0$ in Line 6. X is corrupt but equal to X' in Line 9, and consequently the error is propagated.

The next step is to apply the checks in the PASC hardening algorithm [23], which we show in Figure 1c. With this approach, we first execute all instructions using the original variables (*i.e.*, X), and then execute all instructions using the replica variables (*i.e.*, X'). During the first execution, the process stores in N the updates to the state variable X (Line 4). In the second execution, it writes directly into X'. Next, the process compares the updates in N and X' and applies them to X. Finally, the process sends a message containing X and its replica X'.

The PASC-like approach detects last-mile corruption scenarios by adding X' to the output message. If a fault corrupts X right before the message is sent in Line 9, the receiver detects it by comparing X and X'. In practice, it is not necessary to send the full value of X' and a CRC is sufficient. Also important, the PASC-like algorithm can detect multi-variable corruptions: the process detects a fault corrupting X and X' between Lines 1 and 2 via the check of Line 5, since X is not modified by Execution 1.

If ECC memory is available, the hardware can perform the checks of Lines 1 and 5. Nonetheless, PASC always performs $check(v)$ comparisons in software. Having duplicated state variables for comparison doubles the memory footprint of the process state.

SEI leverages the presence of ECC to minimize memory overhead, but it requires a different algorithm to deal with the fact that a variable and its replica are not stored separately (see Figure 1d). During the first execution, SEI takes a snapshot O of the original value of X. The new reset phase restores the snapshot and stores the state updates in N. The second execution is then executed again on the original state. This snapshot-and-reset strategy introduces additional hardening-specific data structures, which could be corrupted by ASC faults. We discuss next how to deal with these corruptions. Hardening of multithreaded applications, another major improvement of SEI over PASC, is also covered in Section 4.2.

4.2 SEI algorithm description

We now present the SEI algorithm and informally argue its correctness in the presence of some ASC faults. For a more detailed description and a complete correctness proof, we refer to our technical report [11].

Overview. SEI modifies all the three phases executed by event-driven threads (see Section 3). Hardening the dispatching and output phases consists of attaching a message replica c to every message m, *e.g.*, in the form of a CRC. The core challenge is hardening the event handling phase. SEI replaces the original event handler with a *hardened event handler* consisting of five phases: an initialization phase (I), a first-execution phase (E1), a reset phase (R), a second-execution phase (E2), and a final validation phase (V) (see Figure 2).

Figure 3 describes these phases in more detail. The operations in Phases I, R, and V are under the control of SEI and independent of the event handler code. Phases E1 and E2 execute the original event handling code of the application; SEI only inserts additional checks and operations according to Rules R1-R9. Some rules are actions taken before or after statements of the original

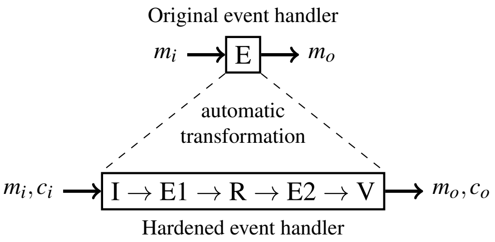


Figure 2: SEI transformation of an event handler E.

handler (denoted using *before/do* and *after/do* constructions), others are statements of the original handler that are replaced (denoted using *replace/with* constructions).

Phase I initializes SEI-internal data structures if the input message m_i matches its CRC c_i, otherwise it discards m_i. Phase E1 updates the state and generates one or more output messages m_o. Phase R restores the state prior to the event handler execution E1. Phase E2 updates the state again and generates the CRC c_o sent with each message m_o. Phase V compares the state updates of E1 and E2, aborting the process if they mismatch.

We start by describing SEI for single-threaded applications and then extend the description to include our multithreading support.

Internal data structures. During Phase E1, SEI takes a snapshot of the current value of variables before they are updated for the first time and stores them in a *snapshot buffer O* (Rule R2). Before Phase E2, it runs a reset phase (Phase R), which stores the current state of the variables updated by Phase E1 in a *new-value buffer N*, and restores the original values in *O*. The new value buffer is compared during Phase V with the *update buffer U*, which is created during Phase E2 (Rule R6), to make sure that the same set of variables is modified.

ASC faults during event handling. SEI executes event handlers twice to detect computation errors. To give some intuition on how SEI detects faults, say a process executes all phases in order I → E1 → R → E2 → V, and the internal data structures used by SEI for hardening are not corrupt. By the fault frequency property (Section 3), we have at most one fault during these steps, so either Phase E1 or E2 is fault-free. The integrity checks of Rules R1 and R5 guarantee that a fault-free execution of E1 or E2 does not read values corrupted by a fault.

Although SEI does not require ECC, an implementation of SEI can leverage ECC in the memory hierarchy because R1 (and its counterpart R5 in Phase E2) can be efficiently checked by the hardware (we perform the check when reading the variable). If the thread reaches Phase V and executes it correctly, then the latest state updates and output messages are correct. Note that even if a fault corrupts one of the two event handlers and skips Phase V, the recipient of the message can still use m_o and c_o to verify that the outputs of $E1$ and $E2$ match.

```
Phase I
if CRC($m_i$) $\neq c_i$ then
    discard $m_i$ and return
initialize SEI variables
increment checking barrier

Phase R
store current values of updated variables in $N$
restore original values from $O$ using $write(v,val)$

Phase E1 (generate output $m_o$)
R1: before $read(v)$ do
        if $\neg check(v)$ then Abort
R2: replace $write(v,val)$ with
        if first write to $v$ then
            add old value of $v$ to $O$
        $write(v,val)$
R3: before acquire lock $L$ do
        add $L$ to $Q$
R4: replace release lock $L$ with
        if not holding $L$ then Abort

Phase E2 (generate CRC $c_o$ for $m_o$)
R5: before $read(v)$ do
        if $\neg check(v)$ then Abort
R6: replace $write(v,val)$ with
        add $v$ to $U$
        $write(v,val)$
R7: after $write(v,val)$ with $v \in m_o$ do
        if last write to $v$ then
            append $val$ to CRC $c_o$
R8: before acquire lock $L$ do
        add $L$ to $Q$
R9: replace release lock $L$ with
        if not holding $L$ then Abort

Phase V
if $\neg check(U)$ or $\neg check(N)$ or $\neg check(v)$ for each $v$ in $N$ then Abort
if CRC($m_i$) $\neq c_i$ then Abort
if CRC($m_o$) $\neq c_o$ for each output message $m_o$ then Abort
if current state inconsistent with the updates in $N$ then Abort
if $N$ and $U$ do not contain the same set of variables then Abort
release all locks in $Q$
wait at checking barrier
```

Figure 3: Rules and actions for each of the phases of a hardened handler. The pairs *before/do*, *after/do*, and *replace/with* indicate operations of the original event handler that are intercepted. The Rules R2 and R6 are described in more detail in Figure 4.

Control-flow gates. SEI can handle much more complex fault scenarios. Due to control-flow faults, a sequence of instructions may be executed multiple times, in full or in part, or may be skipped altogether. We use *control-flow gates*, similar to PASC, to detect incorrect control-flow jumps from one phase of the hardened event handler (Figure 2) into another. We show a simplified example in Figure 4a, in which we use a control-flow variable *cf* (initially set to FALSE) to detect control-flow faults jumping from some phase P_1 to its subsequent phase P_2 and the other way around. Both phases, and thus Lines 1 and 5, represent multiple instructions. If a fault jumps from some instruction in P_1 to some instruction in P_2, then *cf* is not TRUE at Line 6, causing the process to abort. Likewise, if a fault jumps from some instruction in P_2 to some instruction in P_1, then *cf* is already TRUE at Line 2, causing the process to abort. Our technical report contains a more detailed description of gates and covers many more control flow scenarios, including cases where faults corrupt the control-flow variables.

Corruption of SEI-internal data structures. SEI also tolerates faults corrupting SEI-internal data structures. We now discuss two example scenarios. First, consider

a) Control-flow gate example

```
   // one phase
1  some phase $P_1$
2  if $read(cf) = $ TRUE then
3      Abort
4  $write(cf, $ TRUE$)$
   // another phase
5  subsequent phase $P_2$
6  if $read(cf) = $ FALSE then
7      Abort
```

b) Detailed hardening Rules R2 and R6

```
R2: replace $write(v, val)$ with
 1      if $read(O_c(v)) = $ FALSE then
 2          $write(O_v(v), read(v))$
 3          if $read(O_c(v)) = $ TRUE then Abort
 4          $write(O_c(v), $ TRUE$)$
 5          if $read(v) \neq read(O_v(v))$ then Abort
 6      $write(v, val)$
 7      if $read(O_c(v)) = $ FALSE then Abort
R6: replace $write(v, val)$ with
 1      $write(U(v), $ TRUE$)$
 2      $write(v, val)$
 3      if $read(U(v)) = $ FALSE then Abort
```

Figure 4: Control-flow gates and details of SEI-internals.

the example of a *cross-execution propagation*: if a fault occurs during the first execution (*i.e.*, Phase E1) and a variable update is not recorded in O, then the update will not be reset. As a consequence, the second execution (*i.e.*, Phase E2) could run incorrectly. Figure 4b shows the detailed pseudocode of Rule R2. We split the data structure O in two maps O_v and O_c. For each variable v updated during Phase E1, a variable $O_v(v)$ keeps the old value of v while a variable $O_c(v)$ determines (with a boolean) whether the old value of v is already contained in $O_v(v)$ or not. Technically, $O_v(v)$ and $O_c(v)$ are also variables in the process state. Before updating a variable v, the process stores the current value of v into $O_v(v)$ if O_c does not *contain* v yet, *i.e.*, $read(O_c(v)) = \text{FALSE}$. After updating v, the process checks that now O_c contains v in Line 7. This guarantees that if v is updated in Line 6, then by fault frequency either (a) no fault occurs before the update in the current (hardened) event handler execution, so the original value of v is stored into $O_v(v)$, or (b) no fault occurs after the update in the current (hardened) event handler execution, so the process crashes if it detects the absence of an entry for v in O_c. A similar pattern is used to update variable U (see Rule R6 in Figure 4b) and during Phase R to update the map N (see technical report for details).

As another example, consider a *snapshot buffer corruption*: if a variable v is updated multiple times during Phase E1 and a fault occurs, then a newer value of v could be written into $O_v(v)$ instead of the original value. To deal with this problem, SEI first writes the value of v into $O_v(v)$ in Rule R2 and later marks v as contained (Lines 2 and 4 of Figure 4b). Between these two operations, SEI checks that v is not contained in O_c yet, and aborts the process otherwise. Say v is updated a second time. After the first update of v, $read(O_c(v)) = \text{TRUE}$, otherwise the process would have aborted in Line 7; hence, the condition of Line 1 in the second update of v is false. If $O_v(v)$ is anyway overwritten in Line 2 during the second update, a fault must have changed the control flow to skip Line 1; by the fault frequency assumption, Line 3 executes correctly and aborts the process.

Computation scalability. Threads in a multithreaded application share the memory space of the process, but they can also have a set of private variables (stack and thread-local variables). Concurrently executing threads and sharing variables make single-threaded hardening techniques, like the one of Correia *et al.* [23], unsuitable for multithreaded applications. Consider two threads t_1 and t_2 that access the same set of shared variables. If a fault occurs, thread t_1 could write an incorrect value in a shared variable v. Thread t_2 could then read from v in the first execution of the event handler code without being able to detect the error through integrity check. Thread t_1 could then write an incorrect value into v again just before t_2 reads v in its second execution of the event handler code. The main consequence is that t_2 may experience a situation that, in a single-threaded setting, is equivalent to multiple state corruptions during the same (hardened) event handler execution.

The basic requirement of SEI to prevent this type of situations is that the threads of the original application access shared state only within critical sections protected by locks, as discussed in Section 3. Furthermore, in the presence of multiple locks, threads avoid deadlocks by acquiring and releasing the locks they need according to a predefined total order (or hierarchy).

SEI prevents error propagation across threads through shared state using three techniques: *deferred lock releasing*, *validated locking*, and *checking barrier*. Deferred lock releasing prevents error propagation among threads *as long as no two threads enter the same critical section.* A thread hardened with SEI postpones lock release operations during Phases E1 and E2 (Rules R4 and R9). The thread releases its locks only after Phase V, which guarantees that a thread only reads validated state updates from another thread. Deferred lock releasing results in longer critical sections, but this is not a problem for liveness since we target applications using lock hierarchies to avoid deadlocks.

Validated locking addresses situations when a control-flow error causes two threads to enter the same critical section. SEI ensures that the process crashes before any message is sent in such cases. During Phase E1, SEI records the locks a thread acquires (Rules R3 and R8). When SEI intercepts release operations in Phase E1 or E2, it verifies that the thread actually holds the lock (Rules R4 and R9). If the verification fails, then the process crashes.

Consider threads t_1 and t_2 entering the same critical section C. Let S be the process state when t_2 executes the first operation of C. Since t_1 and t_2 are in the same critical section, there must have been a fault during the execution of the current event handler of t_1 or t_2 before S. By fault frequency, there is no fault after S in the current hardened event handler execution of t_1 or t_2. Given S and C, there

```
while(1) {
  ilen = recv_msg_and_crc(imsg, &crc);
  // hardened event handler
  if (__begin(imsg, ilen, crc)) {
    do_something_here(msg);
    omsg = create_a_message_here(&olen);
    __output_append(omsg, olen);
    __output_done(); // finalize CRC
    __end();
  } else continue; // discard invalid input
  send_msg_and_crc(omsg, olen, __crc_pop());
}
```

Figure 5: Example of event loop and hardened handler.

is a unique set of locks that will be eventually released after C. At the state S, at most one of the two threads holds all necessary locks in the set. While releasing the locks, either t_1 or t_2 detects that it does not hold the necessary locks and crash the process. This verification step is still not sufficient, however. Say t_1 does not hold the necessary locks but t_2 does. Even if t_1 crashes the process upon exiting C, t_2 might still have time to exit C correctly and send an incorrect message before the crash.

The checking barrier is designed to prevent this last problem. It guarantees that if two threads execute a critical section concurrently, like t_1 and t_2 in the previous example, they do not send out any message before both have exited their critical sections and checked their locks. Each thread is associated to a concurrency counter, initially zero. When starting the execution of an event handler, a thread increments its own counter (Phase I). After comparing all state modifications in Phase V, the thread increments its counter again. If the counter value of any thread is odd, then it indicates that the thread (*e.g.*, t_2 in the previous example) might hold locks that have not been yet released and verified. After incrementing its counter, a thread takes a snapshot of the current counters for all threads and waits until all threads have either an even counter (they have released and verified all their locks and are ready to complete the execution of the event handler) or a counter higher than the snapshot (they have already started the next event handler execution). The checking barrier's caveat is the additional assumption that a single fault cannot make a thread skip the barrier increment (Line 4 of Phase I), enter a critical section in Phase E1 or E2 without acquiring a lock, and write an incorrect value onto a variable, because this would create an undetectable error for other threads. This scenario did not arise in our fault injection experiments.

5 SEI-hardening implementation

We now present `libsei`,[1] a library designed to automatically harden crash-tolerant distributed systems. `libsei` does not require re-developing the system from scratch, enabling existing code to be hardened with minimal effort, as we discuss in this section and in Section 6.

Hardening code with `libsei`. Hardening an event handler using `libsei` only requires: (i) marking the beginning and the end of an event handler using the macro functions `__begin()` and `__end()`; (ii) calling `__output_append(var, var_len)` to indicate that a variable `var` is added to the current output messages; (iii) calling `__output_done()` to indicate that the output message is complete and its CRC can be finalized and added to the output buffer; (iv) appending CRCs to output messages after retrieving them by calling `__crc_pop()`; and finally (v) starting the compiler as described below. The developer must enclose all operations modifying the process state with `__begin()` and `__end()`. During run time, the event handler executes twice with mechanism similar to `setjmp/longjmp` [33] implemented in `libsei`. Dispatching and output phases are external to `libsei` and do not require interaction with the library. Note that `__output_append()` and `__output_done()` can be called multiple times to generate multiple output messages in one handler.

Figure 5 shows the pseudo-code of a typical event-based process. The functions provided by `libsei` are prefixed with "__"; all remaining code is part of the pre-existing code base that needs to be hardened. Apart from adding some annotations and adding CRCs to messages, which is good practice anyway, there is not much a developer needs to do for hardening.

When the hardened system runs with multiple threads, the function `__barrier()` returns false if the thread should wait for another thread to complete the execution of its handler. The developer is responsible for calling `__barrier()` and blocking the output while it returns false. In Section 6, we discuss how to mitigate the overhead of blocking on the checking barrier.

Development effort. Scaling to large or existing code bases requires minimizing the development effort of using `libsei`. A major challenge is storing snapshots and state updates transparently. Instead of letting the developer notify the hardening library about state accesses, `libsei` automatically intercepts memory operations using a compiler transformation available out-of-the-box. In particular, we use transactional memory (TM) support of GCC, which is available from version 4.7 [27]. The TM compiler option redirects all memory operations within `__begin()` and `__end()` markers to a standardized application binary interface (ABI) [34]. Note that `libsei` provides the ABI but does not implement or rely on a TM algorithm. `libsei` merely executes procedures that store snapshots, state updates and perform validation, as described in Section 4.

`libsei` allows the developer to choose what event

[1] `http://bitbucket.org/db7/libsei`

handlers are and to protect only the important variables. For example, if a piece of code only manipulates a performance statistics variable, the developer might decide to keep the code outside any event handler since the variable does not contain critical data for the application safety. Also, `libsei` supports local handlers, helping the developer to call event handlers without explicitly receiving a message by marking the event handler with `__begin_nm()`, taking no messages as argument.

`libsei` internals. `libsei` tracks lock acquisitions and memory management by wrapping the `pthread_mutex` interface, `malloc()` and `free()`. After calling these functions in Phase E1, `libsei` saves the arguments and return value of the calls in a queue. In Phase E2, after checking the arguments to be the same as in Phase E1, it returns the values in the queue to the caller. Deallocations, similarly to lock releases, are postponed to the end of Phase V. In general, any function performing an external action – *e.g.*, sending a message – called inside an event handler has to be wrapped since it will be executed twice; the compiler terminates with an error otherwise. Among others, `libsei` currently wraps `sendto()` and `sendmsg()`, postponing their calls until the end of the second execution. No wrapper is necessary for external actions performed outside the event handler.

By default, `libsei` relies on memory error detection codes to keep variable replicas and execute *check* operations. This allows us to nearly eliminate the CPU and memory overhead of these operations.

6 Hardening real-world code bases

We have hardened two applications implemented in C: `memcached` and Deadwood. `memcached` is a popular multithreaded in-memory key-value cache [24], highly optimized for performance, that exposes a `get/set` interface to remote clients. `memcached` is essentially a large hashtable with an LRU eviction logic with linked lists to evict items. Deadwood is the single-threaded recursive DNS resolver of MaraDNS [49]. We have used `memcached` 1.4.15 and Deadwood version 3.2.05.

There are three main steps to harden a code base.

Step 1: Event handlers annotation. The initial challenge is choosing the right code lines to introduce the event handler markers. A good understanding of the code base is necessary to determine what state is persistent across the processing of multiple requests. In `memcached`, we marked 8 event handlers and added 7 lines related to the CRC of messages. More than 120 functions were automatically instrumented. In Deadwood, we marked 2 handlers and added 8 lines of code. More than 170 functions were automatically instrumented.

Step 2: Code base adaptation. Instrumentation is particularly simple in distributed systems that are logically organized as a collection of event handlers. These are common and Deadwood is a good example; we had to adapt only 2 code lines of Deadwood, moving a buffer to the heap to enable the reset of updates. Standard distributed computing algorithms such as the ones for state machine replication are typically specified and implemented as event-based algorithms as well. In some distributed system implementation, however, identifying a clean event-based pattern may be more challenging. Hardening required modifying and adding about 60 code lines to `memcached` because it does not always follow the pattern "dispatching, handling, output". One example is when an event handler of a `get` request retrieves an item: after sending the content of the item back to the client in the output phase, `memcached` decrements the reference counter of the item, which, being part of the state, should also be modified in hardened handlers. For such cases we have used local event handlers (see `__begin_nm()` in Section 5).

Another issue is that SEI currently only supports lock-based synchronization (see Section 4). The slab allocator of `memcached`, for example, uses *ad hoc* synchronization, so we disabled it for the hardened version. We left it enabled for the original version, however.

Step 3: Performance tuning. In some cases, the TM compiler might "over-protect" the code from the SEI's point of view. In Deadwood, dozens of strings are allocated and freed in the scope of a single handler; although these strings are in the heap memory, they are local variables of the handler and do not have to be protected. The developer can inform `libsei` to ignore writes into a region of memory, *e.g.*, into a string, by calling `__ignore_addr(addr, size)`. Moreover, if a complete function only modifies local variables, the instrumentation of the function can be disabled by declaring it with the `SEI_LOCAL` attribute.

To mitigate the effect of the checking barrier on the system scalability, the developer can adapt the system to handle other requests while a thread is waiting for other threads to complete the execution of concurrent event handlers. In `memcached`, a thread always serves another connection if sending a message would block the thread on the socket. We consequently fake a "would-block" case when a thread has to wait for the barrier. The caveat of this solution is the further 40 lines of code added to `memcached`. Alternatively, one can disable the barrier altogether, allowing threads to complete the handler execution without waiting for other threads. This solution requires no additional code change, but assumes locks cannot be skipped by ASC faults. We have implemented and evaluated both approaches and report results next.

Group	Fault	Description
CF	CF	IP register changes (control-flow fault)
DF	WREG	register value changes after it is written
	WVAL	memory value changes after it is written
	WADDR	calculated address changes before write
	RADDR	calculated address changes before read
RD	RREG	register value changes before it is read
	RVAL	memory value changes before it is read

Table 1: Fault types supported by our tool.

7 Fault coverage evaluation

The evaluation of SEI comprises two parts: fault coverage and performance. In this section, we report our fault coverage results; the performance results appear in Section 8. In the interest of space, we focus in this paper on the memcached results and briefly mention Deadwood experiments only to reinforce these results. Our technical report [11] contains all results for Deadwood.

To assess fault coverage, we performed two groups of experiments. First, we perform an extensive software fault injection campaign. Our goal is to determine (1) whether SEI effectively guarantees error isolation; and (2) how memory and computation scalability affect fault coverage. The second part consists of hardware fault injection using the dynamic voltage scaling of a processor. Our goal here is to collect evidence that our approach can indeed detect and isolate real, physically induced faults.

7.1 Software fault injection

Setup and methodology. In our fault injection experiments, we follow the approach of Basile *et al.* [8] and Correia *et al.* [23] injecting single bit flips. We have implemented a tool with Intel's Pin dynamic binary instrumentation framework [41] to inject faults during runtime. Our tool can inject three groups of faults described in Table 1. A control-flow (CF) fault flips a bit of the instruction pointer. A fault in the data-flow (DF) group affects the computation: WREG and WVAL represent incorrectly computed values that are respectively written into a register or a memory location, *e.g.*, an addition that results in a wrong value and is stored in a register; WADDR and RADDR represent computational errors while calculating an indexed address for reading or writing from memory. Finally, a fault in the RD group directly corrupts a register (RREG) before being used or a memory location (RVAL) before being read.

Field studies show that most memory faults are detected by ECC [32, 50]. Injected RVAL faults, however, automatically overwrite both, the value and its ECC. Hence, RVAL faults represent worst-case scenarios in which the ECC memory is not able to detect data corruption as assumed by corruption coverage (see Section 3).

Group	Variant	Undetected	Det/SEI	Det/other	Total
CF	mc	9.66%	-	90.34%	6690
	mc-sei	0.06%	14.70%	85.23%	6515
	mc-sei-dup	0.00%	9.87%	90.13%	6594
DF	mc	44.18%	-	55.82%	15180
	mc-sei	0.15%	57.55%	42.29%	20264
	mc-sei-dup	0.00%	45.81%	54.19%	15991
RD	mc	33.04%	-	66.95%	10614
	mc-sei	0.52%	46.78%	52.70%	11508
	mc-sei-dup	0.00%	49.13%	50.87%	11442

Table 2: Errors classified in undetected, SEI-detected, and detected with other mechanisms. Total errors out of 8,000 executions for each fault-variant combination.

To speed up our experiments and make the results reproducible, we have modified memcached to read commands from an input-trace file and write responses into an output-trace file by wrapping functions reading from and writing to sockets. To compare the output trace, we first create a golden run output-trace file. We perform two sets of experiments. The first set studies the fault coverage of SEI and the effects of leveraging hardware error detection codes in the implementation. We run, with a single thread, the unhardened memcached (mc), the SEI-hardened variant (mc-sei) with hardware error detection codes, and a further SEI-hardened variant (mc-sei-dup) with duplicated state assuming no error detection codes in hardware. The second set of experiments investigates whether the computational scalability aspect of our implementation affects the fault coverage. In this set, we run mc-sei and mc-seil with 4 threads; mc-seil has the checking barrier disabled and assumes that locks are not skipped. We perform 8,000 executions for each fault type and each single-threaded variant, with a subtotal of 64,000 executions for the DF group, 24,000 for the RD group, and a total of 168,000 executions (see Table 2). For the multithreaded experiments, we perform a total of 80,000 executions (see Table 3).

In each run, one fault is injected at a randomly selected instruction inside or outside the event handler including shared libraries; Pin cannot, however, instrument instructions inside syscalls. A fault that causes a trace deviation, *e.g.*, an unexpected message or a shorter trace, produces a *manifested error*. The errors we report are all manifested, consequently we refer to them as just errors henceforth.

Fault coverage and memory scalability. We initially experimented with a single thread to observe the effects of faults without the effects of concurrent access. Table 2 summarizes the results of our fault injection experiments with a single thread. The right-most column shows, for each fault-variant combination, the total number of errors out of the total of each group. Errors are detected or undetected, shown as percentage of the total number of errors. *Undetected* errors are *corrupt* output messages that cannot be detected by the client. They correspond to error propagation scenarios where the error isolation property is violated. Detected errors are further divided into *det/SEI*, *i.e.*, errors detected and isolated by libsei, for example, crashes initiated by the library or invalid messages detectable at the client; and *det/other* errors, *i.e.*, errors detected or isolated by other mechanisms, for example, crashes due to segmentation fault or assertions, infinite loops, and also error messages or partial messages detectable at the client. Note that each error propagation percentage is relative to the total observations in each fault group. They do not express probabilities since the real frequency of CF, DF, and RD faults is unknown.

Hardening memcached drastically decreases the undetected errors. The native mc variant shows from 9% up to 44% undetected errors, depending on the fault group. In contrast, the mc-sei variant shows at most 0.15% undetected errors for DF faults and 0.52% for RD faults. The latter result indicates SEI is also resilient to fault scenarios where the ECC memory does not detect data corruption. Hardening Deadwood shows similar trends, reducing undetected errors, for example, from 32.38% down to at most 0.12% for the DF group.

Like PASC [23], mc-sei-dup uses software-duplicated state and detects all injected faults. As this work focuses on the use of hardware error detection, we now analyze how errors manifest specifically on the mc-sei variant.

Detected non-silent errors. Since hardening cannot guarantee fail-stop behavior, some errors are non-silent: clients perceive them as unexpected messages. A message is *invalid* if the message CRC does not match the message payload. From 0.7% up to 3.4% of the errors in mc-sei are invalid messages, representing the majority of non-silent errors. Some messages also arrive truncated at the client, *e.g.*, when memcached crashes before writing the complete message out. Interestingly, memcached itself produces error messages, for example, when a fault makes memcached think it is out of memory. Truncated and error messages constitute up to 1.2% of the errors. As shown in Table 2, undetected errors (*i.e.*, corrupt but valid messages) represent up to 0.52% of the errors and are the only cases that can violate error isolation. We now study these cases in detail.

Undetected errors analysis. Analyzing the log files of our experiments, we identified pointer corruption as the major source of undetected errors in mc-sei. Leveraging hardware error detection, as SEI does, has the side effect that variables and their replicas (the ECC data) are stored in the same memory location and accessed together by the processor. A fault corrupting a pointer to a variable in an undetectable manner causes both, the variable and its replica, to become corrupt, invalidating the fault diversity assumption. It is consequently a type of fault not covered by our fault model. The very low overhead of libsei and the results presented above (a drop of undetected errors from 44.18% to only 0.15%) are encouraging, however. Note also that using software replication overcomes this problem because we use two separate pointers for the value and its replica. Using software replication constitutes a trade-off between memory footprint and fault coverage (see mc-sei-dup in Table 2).

Group	Variant	Undetected	Det/SEI	Det/other	Total
CF	mc-sei	0.02%	13.78%	86.20%	6366
	mc-seil	0.05%	12.84%	87.11%	6330
DF	mc-sei	0.16%	58.58%	41.26%	19484
	mc-seil	0.28%	58.61%	41.11%	19088

Table 3: Errors for 4-threaded executions classified in undetected, detected with SEI, and detected with other mechanisms. Total errors out of 8,000 executions for each fault-variant combination.

To understand a typical scenario of error propagation, consider the following instructions, which are executed upon completion of sending a reply:

```
// item *it = *(c->icurr);
mov    (%rax),%rax
mov    %rax,-0xe0(%rbp)
```

After replying to a get request, memcached decrements the reference counter of the retrieved item. The object c is the connection, and *(c->icurr) is the address of the retrieved item, which is kept in the hashtable. The first instruction stores the address of the current item, *(c->icurr), into register rax. The second instruction moves the address into the stack, *i.e.*, into the target address -0xe0(%rbp). In our logs, a WADDR fault flipped the calculated address, making the mov operation write the pointer just after the stack. The execution proceeded to decrement the reference counter, which is executed in a hardened local event handler. The pointer used, however, was the wrong pointer because the address -0xe0(%rbp) still pointed to an old item in the hashtable. The old item had its reference counter decremented and was freed since its reference counter reached zero. The memory location was later reused for another entry of the hashtable, resulting in two item entries (keys) pointing to the same item object in memory incorrectly.

Computational scalability effects. Table 3 shows the results for our multithreaded experiments. The results indicate that (1) multithreaded executions do not present more undetected errors than single-threaded executions; and (2) although mc-seil assumes locks cannot be skipped, it does not show substantially more undetected errors than mc-sei. In particular, CF faults, which can potentially jump over locks, resulted in less than 0.1% of undetected errors in both variants.

Variant	Undetected	Det/SEI	Det/other	Total
mc-sei	0	11	457	468
mc	4	-	464	468

Table 4: Errors when undervolting CPU.

7.2 Hardware fault injection

Software fault injection can reproduce fault cases very precisely, making it easier to analyze and understand failures. However, there is the risk of introducing a bias, and consequently, we have also used hardware fault injection to reproduce realistic and unbiased failure scenarios.

We perform hardware fault injection by using the dynamic voltage and frequency scaling (DVFS) support of an AMD FX based multi-core CPU (Bulldozer). DVFS can be used to undervolt cores, reducing the voltage below the predefined value while keeping the frequency constant. The scenario of our experiments could be the effect of misconfiguration of power-saving options or of a power supply failure. Note that future microprocessors are also expected to run at lower voltage, thus increasing the likelihood of data corruption [15].

We experimented with variants mc and mc-sei running a single thread. After launching the application, we lowered the voltage between 100 and 150 mV of the nominal CPU voltage (1.225 V). Table 4 shows the outcome of 936 observations. The application often crashed in at most 40 seconds of execution. In addition to crashing, the machine froze very often, explaining the reduced number of experiments performed.

The vast majority of errors were crashes caused by segmentation faults, invalid instruction errors, and other errors detected by the operating system. In mc-sei, 2.35% of the errors (11 cases) were detected by `libsei`, and we observed no undetected errors. In mc, 0.85% of the errors (4 cases) were undetected. Although not conclusive, the experiment indicates that (1) undetected errors, *i.e.*, corrupt messages, can happen due to hardware faults; and (2) some of these faults manifest as ASC faults and are successfully detected by `libsei`.

8 Performance evaluation

Setup and methodology. We run the `memcached` with a hashtable of 2 GB on a 12-core 2.66 GHz Intel Xeon X5650 machine (Linux 3.8 kernel). We use 8 client machines with a similar configuration (8-core 2 GHz Xeon) connected via Gigabit Ethernet. Each client machine runs one instance of Facebook's `mcblaster` workload generator [38]. Each `mcblaster` instance measures averages of the throughput and response time for 60 s.

The workload can be configured with *value size* in bytes. One client machine with 64 connections is started for each `memcached` *thread*. Clients randomly select (using uniform distribution) the next key to be issued from the integer set $\{1,\ldots,K\}$ where K is called *key range*. Finally, the *load* is the aggregated number of requests per second issued by all clients. Clients mainly issue `get` requests since they represent the vast majority of operations in typical workloads [5, 45, 52].

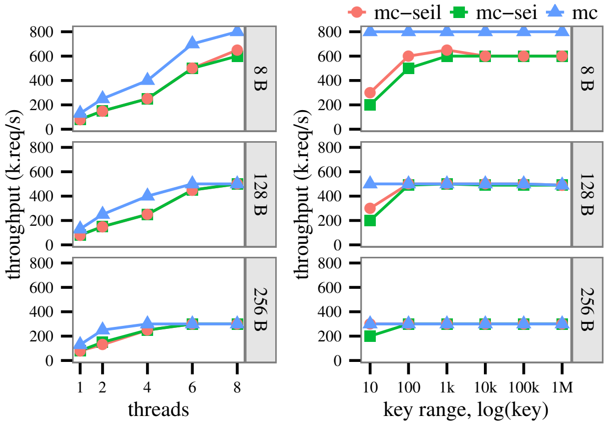


Figure 6: Throughput of get requests varying threads (with key range of 1000) and key range length (with 8 threads) for different value sizes (8, 128 and 256 bytes)

We consider the following `memcached` variants: mc, mc-sei and mc-seil. Stock `memcached` has an important bottleneck due to a global lock protecting the LRU eviction list, *i.e.*, `cache_lock`, which is known to affect scalability [5]. We have improved this bottleneck to increase scalability by having all our variants of `memcached` acquiring the `cache_lock` with `trylock()`, and only updating the list if there are no concurrent updates. Even with this bottleneck improvement, mc still does not scale above 8 threads, so we limit our experiments to up to 8 threads. Finally, to avoid modifying the workload generator, the hardened `memcached` variants compute 32-bit CRCs as prescribed by the algorithm, but do not send them along with messages. The expected performance impact of 4 bytes of CRC is negligible when added.

Deadwood is a single-threaded server. It follows a similar setup, but with up to 20 client machines running `nsping` to query the IP of 100 popular websites.

Computation and memory scalability. Figure 6 (left) represents the scalability limit for `memcached` when varying the number of threads from 1 to 8. The y-axis depicts the maximal throughput that can be achieved while keeping the average response time across all requests below 1 ms, a realistic response time target for `memcached`. We also vary the value size from extremely small messages (8 B) to medium messages (256 B). With larger value sizes, fewer threads are necessary to achieve the maximal throughput with any variant; all variants achieve their maximal throughput with 8 threads. With value sizes larger or equal to 128 B and 8 threads, mc-sei and mc-seil show negligible throughput overhead. With a value size of 8 B and 8 threads, the overhead is 25% for mc-sei and 20% for mc-seil. mc-seil shows a lower overhead than mc-sei due to the disabled SEI's barrier.

Figure 6 (right) depicts the maximal throughput achieved when varying the key range and the value size with 8 threads. Few keys introduce contention between the threads, since they access the same buckets, acquiring at least 2 locks per request. Critical sections become longer due to hardening. Consequently, the scalability with queries spanning very few keys, *e.g.*, 10 keys, is limited. Such scenario could also represent a workload with a few hot keys. We expect, however, a memcached instance to host and serve many thousands of different keys. As we distribute the workload across more keys, there is less contention and consequently more opportunity for concurrent execution. The overhead with 1 M keys and 8 B value sizes, for example, is about 25% for both mc-sei and mc-seil. The overhead becomes negligible with larger value sizes and more than 100 keys.

Regarding memory overhead, each thread requires about 30 KiB for hardening-related data structures.

Single-thread scenarios. libsei is designed to amortize its overhead with the number of threads. Multi-threading can release pressure on the CPU, making it more likely for the system to become network bound. We now consider single-threaded scenarios with Deadwood and memcached running with one thread – see [11] for details on these experiments.

When the system is not overloaded, the response time overhead of mc-sei is small. For example, at a load of 10 k.req/s, the difference of response time between mc and mc-sei varies from 2% to 7% depending on the size of the messages – we experimented with values of 8 B up to 8 KiB. In Deadwood, depending on the response size, hardening incurs an overhead from 13% up to 21% in the response time for loads of 1 k.req/s.

Deadwood becomes CPU bound very quickly. As a result, the throughput overhead under maximal request load reaches 40% to 50%. Using the timestamp counter of the processor, we measured the average number of cycles Deadwood consumes for a single request in the dispatch, handling, and output phases (averaged over 10,000 requests). Table 5 shows the percentage of CPU cycles spent in each phase relative to the native variant. The dispatch and output phases do not increase significantly with hardening. The hardened handling phase takes, however, 2.4 times the number of the cycles of the native counterpart. This overhead is caused by the double execution of the event handler, by the code instrumentation, and by the checks in libsei. Since the duplicated part constitutes only 27% of the used cycles, the cycle-overhead of processing a single message by hardened Deadwood is only 41%.

Variant	Dispatch	Handling	Output	Total	Cycles
Native	41.75%	27.51%	30.74%	100%	83 k
SEI-hardened	42.71%	66.81%	31.47%	141%	117 k
Overhead	+0.94%	+39.30%	+0.73%	+41%	

Table 5: Average CPU cycles consumed for a single request relative to the native Deadwood variant.

In contrast, mc-sei is not likely to saturate even with a single thread. For 1 KiB large values, mc-sei has an overhead of 20%. For 4 KiB or larger values, both mc-sei and mc are network bound and show no significant difference in the throughput.

Overall, even in single-threaded scenarios we observed no more than 50% overhead. The low overhead is due to the hardening of application event handlers, but not the underlying software components, such as the operating system. SEI expects faults in these components to manifest as ASC faults, corrupting the application state or its messages. According to our fault injection experiments, SEI is sufficient and a "duplicate everything" strategy is not strictly necessary. We expect long event-handling phases, however, to induce higher overheads.

9 Conclusion

We have proposed a novel algorithm for ASC hardening, SEI, that can leverage mechanisms provided in hardware, such as error correction codes in memory modules, to minimize overhead. The exercise of hardening an existing system like a DNS server and memcached exposed a number of challenges, mostly related to deviations to the structure our algorithm expects. Yet, we were able to harden it with some reasonable amount of effort. SEI introduces a negligible overhead in applications that are not CPU-bound and is effective in avoiding error propagation. The residual error propagation observed in our fault injection results is due to pointer corruption, which SEI is vulnerable to when using hardware ECC. It is subject of future work to design new techniques or extensions that are able to overcome this limitation while using ECC memory for systems that are potentially more susceptible to pointer corruption.

Acknowledgments

This work was partially supported by the state of Saxony under grant of ESF 100111037 and by the German Research Foundation (DFG) within the Cluster of Excellence "Center for Advancing Electronics Dresden". We would like to thank our reviewers and our shepherd Richard Mortier for the useful feedback.

References

[1] AMAZON. Amazon S3 availability event: July 20, 2008. http://status.aws.amazon.com/s3-20080720.html, July 2008.

[2] AMAZON. New defective S3 load balancer corrupts relayed messages. https://forums.aws.amazon.com/thread.jspa?threadID=22709, June 2008.

[3] AMAZON. Odd data corruption during download. https://forums.aws.amazon.com/thread.jspa?messageID=86214, Apr. 2008.

[4] AMAZON. Single-bit corruption of a small percentage of S3 data. https://forums.aws.amazon.com/thread.jspa?messageID=262676, July 2011.

[5] ATIKOGLU, B., XU, Y., FRACHTENBERG, E., JIANG, S., AND PALECZNY, M. Workload analysis of a large-scale key-value store. In *Proceedings of the 12th ACM SIGMETRICS/PERFORMANCE joint international conference on Measurement and Modeling of Computer Systems* (New York, NY, USA, 2012), SIGMETRICS '12, ACM, pp. 53–64.

[6] BAKER, J., BOND, C., CORBETT, J. C., FURMAN, J. J., KHORLIN, A., LARSON, J., JEAN, LÉON, M., LI, Y., LLOYD, A., AND YUSHPRAKH, V. Megastore: Providing scalable, highly available storage for interactive services. In *5th Biennial Conference on Innovative Data Systems Research (CIDR)* (2011).

[7] BARTLETT, W., AND SPAINHOWER, L. Commercial fault tolerance: A tale of two systems. *IEEE Transactions on Dependable and Secure Computing 1*, 1 (2004), 87–96.

[8] BASILE, C., LONG, W., KALBARCZYK, Z., AND IYER, R. Group communication protocols under errors. In *Proceedings of the 22nd IEEE Symposium on Reliable Distributed Systems* (2003), pp. 35–44.

[9] BEHRENS, D., FETZER, C., JUNQUEIRA, F. P., AND SERAFINI, M. Towards transparent hardening of distributed systems. In *Proceedings of the 9th Workshop on Hot Topics in Dependable Systems* (2013), HotDep'13.

[10] BEHRENS, D., KUVAISKII, D., AND FETZER, C. HardPaxos: Replication Hardened against Hardware Errors. In *IEEE 33rd International Symposium on Reliable Distributed Systems (SRDS), 2014* (Oct. 2014), pp. 232–241.

[11] BEHRENS, D., SERAFINI, M., ARNAUTOV, S., JUNQUEIRA, F., AND FETZER, C. Scalable error isolation. Tech. Rep. TUD-FI15-01-Februar 2015, ISSN 1430-211X, Technische Universität Dresden, Fakultät Informatik, Feb. 2015. http://bitbucket.org/db7/libsei.

[12] BEHRENS, D., WEIGERT, S., AND FETZER, C. Automatically tolerating arbitrary faults in non-malicious settings. In *Proceedings of the Sixth Latin-American Symposium on Dependable Computing (LADC)* (April 2013), pp. 114–123.

[13] BHATOTIA, P., WIEDER, A., RODRIGUES, R., JUNQUEIRA, F., AND REED, B. Reliable data-center scale computations. In *Proceedings of the 4th International Workshop on Large Scale Distributed Systems and Middleware* (New York, NY, USA, 2010), LADIS '10, ACM, pp. 1–6.

[14] BORKAR, S. Designing reliable systems from unreliable components: The challenges of transistor variability and degradation. *IEEE Micro 25*, 6 (2005), 10–16.

[15] BORKAR, S., AND CHIEN, A. A. The future of microprocessors. *Communications of the ACM 54*, 5 (2011), 67–77.

[16] BORKAR, S., ET AL. Microarchitecture and design challenges for gigascale integration. In *MICRO* (2004), vol. 37, pp. 3–3.

[17] BURROWS, M. The chubby lock service for loosely-coupled distributed systems. In *OSDI '06: Proceedings of the 7th symposium on Operating systems design and implementation* (Berkeley, CA, USA, 2006), USENIX Association, pp. 335–350.

[18] CASTRO, M., AND LISKOV, B. Practical Byzantine fault tolerance and proactive recovery. *ACM Transactions on Computer Systems 20*, 4 (2002).

[19] CASTRO, M., RODRIGUES, R., AND LISKOV, B. BASE: Using abstraction to improve fault tolerance. *ACM Transactions Computer Systems 21*, 3 (Aug. 2003), 236–269.

[20] CHANDRA, T. D., GRIESEMER, R., AND REDSTONE, J. Paxos made live: an engineering perspective. In *Proceedings of the twenty-sixth annual ACM symposium on Principles of distributed computing* (New York, NY, USA, 2007), PODC '07, ACM, pp. 398–407.

[21] CONSTANTINESCU, C. Trends and challenges in vlsi circuit reliability. *IEEE Micro 23*, 4 (2003), 14–19.

[22] CORBETT *et al.*. Spanner: Google's globally-distributed database. In *Proceedings of the 10th USENIX conference on Operating Systems Design and Implementation* (Berkeley, CA, USA, 2012), OSDI'12, USENIX Association, pp. 251–264.

[23] CORREIA, M., FERRO, D. G., JUNQUEIRA, F., AND SERAFINI, M. Practical hardening of crash-tolerant systems. In *2012 USENIX Annual Technical Conference* (2012).

[24] DANGA INTERACTIVE, INC. memcached – a distributed memory object caching system. http://memcached.org.

[25] DINABURG, A. Bitsquatting: DNS hijacking without exploitation. In *Defcon 19* (2011).

[26] FENG, S., GUPTA, S., ANSARI, A., AND MAHLKE, S. Shoestring: Probabilistic soft error reliability on the cheap. In *Proceedings of the Fifteenth Edition of ASPLOS on Architectural Support for Programming Languages and Operating Systems* (New York, NY, USA, 2010), ASPLOS XV, ACM, pp. 385–396.

[27] Transactional Memory in GCC. http://gcc.gnu.org/wiki/TransactionalMemory.

[28] GUPTA, A., YANG, F., GOVIG, J., KIRSCH, A., CHAN, K., LAI, K., WU, S., DHOOT, S., KUMAR, A., AGIWAL, A., BHANSALI, S., HONG, M., CAMERON, J., SIDDIQI, M., JONES, D., SHUTE, J., GUBAREV, A., VENKATARAMAN, S., AND AGRAWAL, D. Mesa: Geo-replicated, near real-time, scalable data warehousing. In *VLDB* (2014).

[29] HAMILTON, M. *Software development: building reliable systems*. Prentice Hall Professional, 1999.

[30] HO, C., VAN RENESSE, R., BICKFORD, M., AND DOLEV, D. Nysiad: Practical protocol transformation to tolerate byzantine failures. In *NSDI'07: Proceedings of the 4th USENIX Symposium on Networked Systems Design and Implementation* (2007).

[31] HUNT, P., KONAR, M., JUNQUEIRA, F. P., AND REED, B. ZooKeeper: Wait-free coordination for internet-scale systems. In *Proceedings of USENIX Annual Technical Conference* (2010).

[32] HWANG, A. A., STEFANOVICI, I., AND SCHROEDER, B. Cosmic rays don't strike twice: Understanding the nature of DRAM errors and the implications for system design. In *ASPLOS* (2012).

[33] IEEE Std 1003.1 and The Open Group Base Specifications, Issue 7. http://pubs.opengroup.org/onlinepubs/9699919799/functions/setjmp.html, 2013.

[34] INTEL. Intel Transactional Memory Compiler and Runtime Application Binary Interface, 2009.

[35] KAPRITSOS, M., WANG, Y., QUEMA, V., CLEMENT, A., ALVISI, L., DAHLIN, M., ET AL. All about eve: Execute-verify replication for multi-core servers. In *OSDI* (2012), vol. 12, pp. 237–250.

[36] KERNEL BUG TRACKER. Data corruption with Opteron CPUs and Nvidia chipsets. https://bugzilla.kernel.org/show_bug.cgi?id=7768, Jan. 2007.

[37] KOTLA, R., ALVISI, L., DAHLIN, M., CLEMENT, A., AND WONG, E. Zyzzyva: Speculative Byzantine fault tolerance. In *Proceedings of ACM SOSP* (2007), pp. 45–58.

[38] KWIATKOWSKI, M. mcblaster - load generator for memcached. http://github.com/fbmarc.

[39] LAMPORT, L. The part-time parliament. *ACM Transactions on Computing Systems (TOCS) 16*, 2 (1998), 133–169.

[40] LAMPORT, L., SHOSTAK, R., AND PEASE, M. The byzantine generals problem. *ACM Transactions on Programming Languages and Systems 4*, 3 (1982).

[41] LUK, C.-K., COHN, R., MUTH, R., PATIL, H., KLAUSER, A., LOWNEY, G., WALLACE, S., REDDI, V. J., AND HAZELWOOD, K. Pin: building customized program analysis tools with dynamic instrumentation. In *Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation* (New York, NY, USA, 2005), PLDI '05, ACM, pp. 190–200.

[42] MERIDETH, M. G., IYENGAR, A., MIKALSEN, T., TAI, S., ROUVELLOU, I., AND NARASIMHAN, P. Thema: Byzantine-fault-tolerant middleware for Web-service applications. In *Proceedings of the 24th IEEE Symposium on Reliable Distributed Systems* (Washington, DC, USA, 2005), SRDS '05, IEEE Computer Society, pp. 131–142.

[43] NIGHTINGALE, E. B., DOUCEUR, J. R., AND ORGOVAN, V. Cycles, cells and platters: an empirical analysis of hardware failures on a million consumer PCs. In *Proc. of Eurosys* (2011), pp. 343–356.

[44] Nikiforakis, N., Van Acker, S., Meert, W., Desmet, L., Piessens, F., and Joosen, W. Bitsquatting: exploiting bit-flips for fun, or profit? In *Proceedings of the 22nd international conference on World Wide Web* (2013), International World Wide Web Conferences Steering Committee, pp. 989–998.

[45] Nishtala, R., Fugal, H., Grimm, S., Kwiatkowski, M., Lee, H., Li, H. C., McElroy, R., Paleczny, M., Peek, D., Saab, P., Stafford, D., Tung, T., and Venkataramani, V. Scaling memcache at facebook. In *Proceedings of the 10th USENIX conference on Networked Systems Design and Implementation* (Berkeley, CA, USA, 2013), nsdi'13, USENIX Association, pp. 385–398.

[46] Oh, N., Shirvani, P., and McCluskey, E. Error detection by duplicated instructions in superscalar processors. *Reliability, IEEE Transactions on 51*, 1 (Mar. 2002), 63–75.

[47] Perry, F., Mackey, L., Reis, G. A., Ligatti, J., August, D. I., and Walker, D. Fault-tolerant typed assembly language. In *ACM SIGPLAN Notices* (2007), vol. 42, ACM, pp. 42–53.

[48] Reis, G., Chang, J., Vachharajani, N., Rangan, R., and August, D. SWIFT: software implemented fault tolerance. In *Proceedings of the International Symposium on Code Generation and Optimization* (Mar. 2005), pp. 243–254.

[49] Sam Trenholme and others. Deadwood recursive DNS resolver. `http://maradns.samiam.org/deadwood`.

[50] Schroeder, B., Pinheiro, E., and Weber, W.-D. DRAM errors in the wild: a large-scale field study. In *Proceedings of the eleventh international joint conference on Measurement and modeling of computer systems* (New York, NY, USA, 2009), SIGMETRICS '09, ACM, pp. 193–204.

[51] Shivakumar, P., Kistler, M., Keckler, S. W., Burger, D., and Alvisi, L. Modeling the effect of technology trends on the soft error rate of combinational logic. *International Conference on Dependable Systems and Networks* (2002), 389.

[52] Venkataramani, V., Amsden, Z., Bronson, N., Cabrera III, G., Chakka, P., Dimov, P., Ding, H., Ferris, J., Giardullo, A., Hoon, J., Kulkarni, S., Lawrence, N., Marchukov, M., Petrov, D., and Puzar, L. TAO: How facebook serves the social graph. In *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data* (New York, NY, USA, 2012), SIGMOD '12, ACM, pp. 791–792.

[53] Wood, A., Jardine, R., and Bartlett, W. Data integrity in HP NonStop servers. In *Workshop on SELSE* (2006).