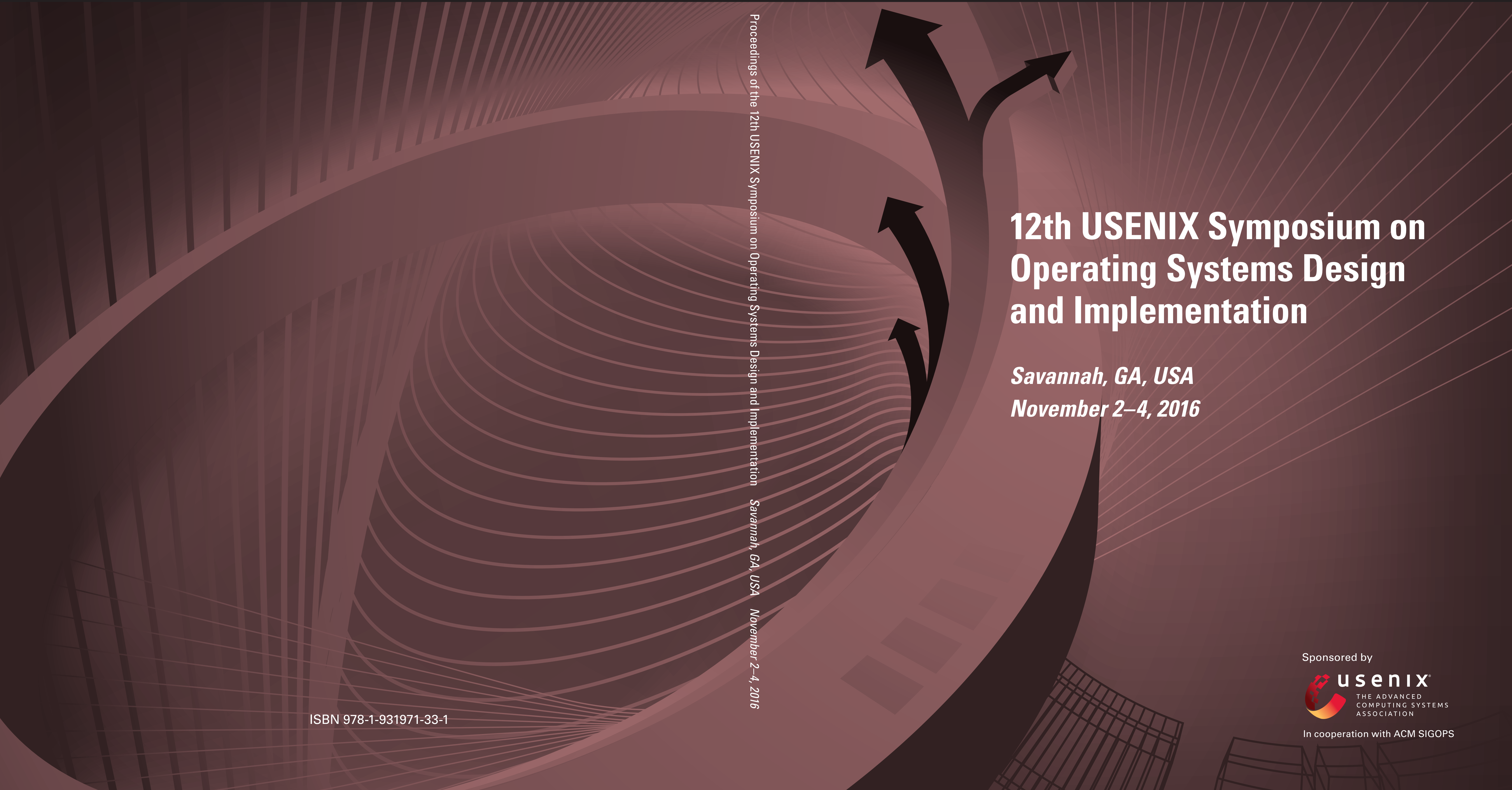


conference

proceedings

12th USENIX Symposium on Operating Systems Design and Implementation

Savannah, GA, USA

November 2–4, 2016

Sponsored by

usenix

THE ADVANCED COMPUTING SYSTEMS ASSOCIATION

In cooperation with ACM SIGOPS

usenix ASSOCIATION

Proceedings of the 12th USENIX Symposium on Operating Systems Design and Implementation *Savannah, GA, USA* *November 2–4, 2016*

ISBN 978-1-931971-33-1

Thanks to Our OSDI ’16 Sponsors

Platinum Sponsor

Google

Diamond Sponsor

Microsoft

Gold Sponsors

HUAWEI NSF

Silver Sponsors

facebook NetApp vmware

Bronze Sponsors

amazon web services Hewlett Packard Enterprise IBM Research

ORACLE Symantec

Industry Partner
Free BSD Foundation

ISBN 978-1-931971-33-1

Thanks to Our USENIX Supporters

USENIX Patrons

Facebook Google Microsoft Research NetApp VMware

USENIX Benefactors

Admin *Linux Pro Magazine*

USENIX Partners

Booking.com Can Stock Photo

Open Access Publishing Partner

PeerJ

USENIX Association

Proceedings of OSDI '16: 12th USENIX Symposium on Operating Systems Design and Implementation

November 2–4, 2016
Savannah, GA, USA

Symposium Organizers

Program Co-Chairs
Kimberly Keeton, *Hewlett Packard Labs*
Timothy Roscoe, *ETH Zurich*

Program Committee
Nitin Agrawal, *Samsung Labs*
Andrea Arpaci-Dusseau, *University of Wisconsin—Madison*
Sujata Banerjee, *Hewlett Packard Labs*
Andrew Baumann, *Microsoft Research Redmond*
Jeff Chase, *Duke University*
Peter Chen, *University of Michigan*
Byung-Gon Chun, *Seoul National University*
Jeff Dean, *Google*
Amol Deshpande, *University of Maryland*
Peter Druschel, *Max Planck Institute for Software Systems (MPI-SWS)*
Sasha Fedorova, *University of British Columbia*
Bryan Ford, *École Polytechnique Fédérale de Lausanne (EPFL)*
Roxana Geambasu, *Columbia University*
Philip Gibbons, *Carnegie Mellon University*
Saikat Guha, *Microsoft Research India*
Gernot Heiser, *NICTA and University of New South Wales*
Y. Charlie Hu, *Purdue University*
Galen Hunt, *Microsoft Research Redmond*
Rebecca Isaacs, *Google*
Michael Kaminsky, *Intel Labs*
Randy Katz, *University of California, Berkeley*
Dejan Kostic, *KTH Royal Institute of Technology*
Tim Kraska, *Brown University*
Orran Krieger, *Boston University*
Phil Levis, *Stanford University*
Boon Loo, *University of Pennsylvania*
Shan Lu, *University of Chicago*
Petros Maniatis, *Google*
Z. Morley Mao, *University of Michigan*
Frank McSherry, *ETH Zurich*
James Mickens, *Harvard University*
Thomas Moscibroda, *Microsoft Research China*
Dushyanth Narayanan, *Microsoft Research Cambridge*
Sam H. Noh, *UNIST (Ulsan National Institute of Science & Technology)*
KyoungSoo Park, *Korea Advanced Institute of Science and Technology (KAIST)*
Simon Peter, *The University of Texas at Austin*
George Porter, *University of California, San Diego*
Dan Ports, *University of Washington*
Chris Rossbach, *VMware Research and The University of Texas at Austin*
Michael Scott, *University of Rochester*
Margo Seltzer, *Harvard School of Engineering and Applied Sciences and Oracle*
Dawn Song, *University of California, Berkeley*
Michael Stumm, *University of Toronto*
Patrick Traynor, *University of Florida*
Dan Tsafrir, *Technion—Israel Institute of Technology*
Robert Watson, *University of Cambridge*
Ellen Zegura, *Georgia Institute of Technology*
Zheng Zhang, *NYU Shanghai*

Poster Session Co-Chairs
George Porter, *University of California, San Diego*
Chris Rossbach, *VMware Research and The University of Texas at Austin*

Steering Committee
Brad Chen, *Google*
Jason Flinn, *University of Michigan*
Casey Henderson, *USENIX Association*
Hank Levy, *University of Washington*
James Mickens, *Harvard University*
Brian Noble, *University of Michigan*
Margo Seltzer, *Harvard School of Engineering and Applied Sciences and Oracle*
Amin Vahdat, *Google and University of California, San Diego*

External Reviewers

Paarijaat Aditya
Ram Alagappan
Gustavo Alonso
Nadav Amit
Michael Andersen
Arka Bhattacharya
Kevin Butler
Michael Cahill
Joao Carreira
David Chisnall
Eslam Elnikety
Gabe Fierro
Aishwarya Ganesan
Ionel Gog
Hariharan Gopalakrishnan
Tyler Harter
Jun He
Suman Jana
Grant Hernandez
John Kolb
James Litton
Jay Lorch
Daniel Margo
Aastha Mehta
Changwoo Min
Pulkit Misra
Thomas Neumann
Gabriel Parmer
Nathan Pemberton
Thanumalayan Sankaranarayana Pillai
Bradley Reaves
Jennifer Rexford
Felix Schuster
Malte Schwarzkopf
Rijurekha Sen
Vitaly Shmatikov
Julian Shun
Viktor Vafeiadis
Anjo Vahldiek
Kapil Vaswani
Dimitrios Vytiniotis
Zev Weiss
John Wilkes
Neeraja Yadwadkar
Suli Yang

OSDI ’16:
12th USENIX Symposium on Operating Systems Design and Implementation
Savannah, GA, USA

(Wednesday, November 2, continues on next page)

(Friday, November 4, continues on next page)

Message from the OSDI '16 Program Co-Chairs

Dear colleagues,

We are delighted to welcome to you to the 12th USENIX Symposium on Operating Systems Design and Implementation, held in Savannah, GA, USA! This year's program includes a record high 47 papers that represent the strength of our community and cover a wide range of topics, including security, cloud computing, transaction support, storage, networking, formal verification of systems, graph processing, system support for machine learning, programming languages, troubleshooting, and operating systems design and implementation.

We received 260 paper submissions, which we reviewed in multiple rounds. Our program committee consisted of 48 "heavy" and "light" members with a mixture of academic and industrial research and practical experience. Papers received 3 reviews in the first round, and we selected 109 papers to proceed to the second round. Second round papers received a minimum of 3 further reviews. For 42 papers, where opinions were divided or where a paper was particularly specialized, we solicited additional expert reviews in a third round. In total, the PC and external reviewers wrote over 1160 reviews.

The submission process included a response period in which authors could answer reviewer questions and address factual errors in the reviews. Responses had a measurable impact on PC meeting discussions, helping some papers and hurting others. Overall, we believe responses were useful in improving the fairness of the review process and the quality of the selected program.

After about a week of lively online discussion among the full PC, we picked 91 papers for the 26 heavy PC members to discuss at a 1.5-day PC meeting held at the University of Washington in Seattle, WA, USA. All discussed papers received a summary of the PC discussion. The PC's discussion selected 47 papers for presentation at the conference, resulting in an 18% acceptance rate. We conditionally accepted all selected papers pending revisions, and the authors and their shepherds made a significant effort to address the reviewers' comments. Because of the improvements due to the shepherding process, we deferred selection of the award papers until the final camera-ready papers were available.

We wish to thank the many people who contributed to this conference. First and foremost, we are grateful to the authors of all submitted papers for choosing to send work of such high quality to OSDI. Thanks also to the program committee for their many hours of hard work in reviewing and discussing the submissions and in shepherding the accepted papers. We are also grateful to the external reviewers who provided an additional perspective on a few papers. We thank Dan Ports and the staff at the University of Washington for hosting the PC meeting. We thank the USENIX staff, who have been fundamental in organizing OSDI '16. Finally, OSDI wouldn't be what it is without its attendees—thank you for creating and sustaining such a vibrant community!

We hope that you will find the program interesting and inspiring and trust that the conference will provide you with a valuable opportunity to share ideas with other researchers and practitioners from institutions around the world.

Kimberly Keeton, *Hewlett Packard Labs*
Timothy Roscoe, *ETH Zurich*
OSDI '16 Program Co-Chairs

Push-Button Verification of File Systems via Crash Refinement

Helgi Sigurbjarnarson, James Bornholt, Emina Torlak, Xi Wang
University of Washington

Abstract

The file system is an essential operating system component for persisting data on storage devices. Writing bug-free file systems is non-trivial, as they must correctly implement and maintain complex on-disk data structures even in the presence of system crashes and reorderings of disk operations.

This paper presents Yggdrasil, a toolkit for writing file systems with *push-button* verification: Yggdrasil requires no manual annotations or proofs about the implementation code, and it produces a counterexample if there is a bug. Yggdrasil achieves this automation through a novel definition of file system correctness called *crash refinement*, which requires the set of possible disk states produced by an implementation (including states produced by crashes) to be a subset of those allowed by the specification. Crash refinement is amenable to fully automated satisfiability modulo theories (SMT) reasoning, and enables developers to implement file systems in a modular way for verification.

With Yggdrasil, we have implemented and verified the Yxv6 journaling file system, the Ycp file copy utility, and the Ylog persistent log. Our experience shows that the ease of proof and counterexample-based debugging support make Yggdrasil practical for building reliable storage applications.

1 Introduction

File systems are a vital operating system service for user applications to manage and persist data. Their correctness is critical to system reliability; file system corruption can damage files and even render the disk unable to mount [12, 13]. Correctly implementing a file system is difficult [23], due to the need to maintain complex on-disk data structures that must remain consistent in the face of power failures and system crashes. Many bugs have been found in commonly used file systems, and have led to serious data losses [27, 35, 39, 45, 52, 54]. Such bugs are likely to continue proliferating due to the complexity of modern storage stacks [1, 8].

Yggdrasil is a toolkit that helps programmers write file systems and formally verify their correctness in a *push-button* fashion. Yggdrasil asks programmers for three inputs: a specification of the expected behavior, an implementation, and consistency invariants indicating whether a file system image is in a consistent state. It then performs verification to check if the implementation meets the specification. If there is a bug, Yggdrasil produces a counterexample to help identify and fix the cause. If the verification passes, Yggdrasil produces an executable file system. It requires *no* manual annotations or proofs about the implementation code.

A key challenge for push-button file system verification is to minimize the proof burden. One approach to verified file systems is to ask programmers to construct a proof of implementation correctness using an interactive theorem prover such as Coq [11] or Isabelle [33]. Pioneering work in this direction includes COGENT [2, 34], Flashix [16, 47], and FSCQ [7], which are impressive engineering achievements. However, writing proofs requires both a high degree of expertise and a significant time investment. For instance, Amani et al. reported that verifying two operations of the BilbyFs file system took 9.25 person months, writing 13,000 lines of proof for 1,350 lines of code [2]. Verifying the FSCQ file system took Chen et al. 1.5 years; the code size is 10× that of xv6, an unverified file system with similar features [7]. To free programmers from such a proof burden, Yggdrasil provides fully automated reasoning.

Conceptually, showing that a file system is correct involves exploring its behavior along all execution paths and against all possible disk states. In practice, such exhaustive exploration is intractable: file systems operate on massive inputs (e.g., entire disks); their code often has many execution paths; and non-determinism adds even more complexity, since one needs to reason about crashes at arbitrary points during execution and reorderings of writes due to the disk cache. Existing file-system automated reasoning tools [52–54] therefore focus on bug finding rather than verification.

Yggdrasil scales up automated reasoning for verifying file systems with the idea of *crash refinement*, a new definition of file system correctness. Crash refinement captures the notion that even in the presence of non-determinism, such as system crashes and reordering of writes, any disk state produced by a correct implementation must also be producible by the specification (see §3 for a formal definition). This definition is amenable to efficient satisfiability modulo theories (SMT) reasoning, an extension of boolean satisfiability. Yggdrasil formulates file system verification as an SMT problem and invokes a state-of-the-art SMT solver (Z3 [15]) to fully automate the proof process.

SMT reasoning is not, by itself, a push-button solution; building verified file systems also requires careful design. Crash refinement enables programmers to implement file systems by *stacking layers of abstraction*: if an implementation is a crash refinement of an (often much simpler) specification, they are indistinguishable to higher layers. The higher layers can use lower specifications without reasoning about the implementation details. This modular design allows Yggdrasil to verify a file system by exhausting all execution paths within a layer while avoiding path explosion between layers.

In addition, crash refinement enables transparent switching between different implementations that satisfy the same specification. Programmers can use simple data structures for verification, and then refine them to more efficient versions with the same correctness guarantees. Separating logical and physical concerns in this fashion allows Yggdrasil to verify complex, high-performance on-disk data structures.

We have used Yggdrasil to implement and verify Yxv6+sync, a journaling file system that resembles xv6 [14] and FSCQ [7], and Yxv6+group_commit, an optimized variant with relaxed crash consistency [5, 37]. To demonstrate Yggdrasil on a broader set of applications, we have built Ycp, a file copy utility on top of Yxv6; and Ylog, which resembles the persistent log from the Arrakis operating system [36]. We have also built general-purpose "peephole optimizers" [28] for file system code (e.g., removing superfluous disk flushes). We believe that the ease of verification makes Yggdrasil attractive for building verified storage applications.

We have been using the Yxv6 file system, which runs on top of FUSE [17], to self-host Yggdrasil's daily development on Linux. It has passed `fsstress` from the Linux Test Project [26] and the SibylFS POSIX conformance tests [42] (except for incomplete features, such as hard links and extended attributes). We have found its performance to be reasonable: within 10× of ext4's default configuration and 3–150× faster than FSCQ. Yggdrasil focuses on single-threaded systems; verifying concurrent implementations is beyond the scope of this paper.

This paper makes the following contributions:

- a formalization of file system crash refinement that is amenable to fully automated SMT reasoning;
- the Yggdrasil toolkit for building verified file systems through crash refinement; and
- a case study of building the Yxv6 file system and several other storage programs using Yggdrasil.

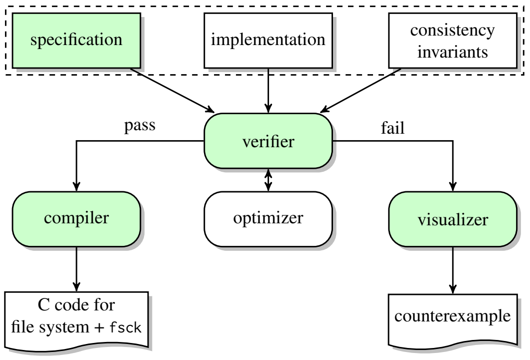


Figure 1: The Yggdrasil development flow. Rectangular boxes (within the dashed frame) denote input written by programmers; rounded boxes denote Yggdrasil's components; and curved boxes denote output. Shaded boxes are trusted to be correct and the rest are untrusted.

The rest of the paper is organized as follows. §2 gives a walkthrough of Yggdrasil's usage. §3 presents formal definitions and the main components. §4 describes the Yxv6 file system and §5 describes other storage applications built using Yggdrasil. §6 discusses Yggdrasil's limitations and our experience. §7 provides implementation details. §8 evaluates correctness and performance. §9 relates Yggdrasil to prior work. §10 concludes.

2 Overview

Figure 1 shows the Yggdrasil development flow. Programmers write the specification, implementation, and consistency invariants all in the same language (a subset of Python in our current prototype; see §3.2). If there is any bug in the implementation or consistency invariants, the verifier generates a counterexample to visualize it. For better run-time performance, Yggdrasil optionally performs optimizations (either built-in or written by developers) and re-verifies the code. Once the verification passes, Yggdrasil emits C code, which is then compiled and linked using a C compiler to produce an executable file system, as well as an `fsck` checker.

This section gives an overview of each of these steps, using a toy file system called YminLFS as a running example. We will show how to specify, implement, verify, and debug it; how to optimize its performance; and how to get a running file system mounted via FUSE [17].

YminLFS is a log-structured file system [44]. It is kept minimal for demonstration purposes: there are no segments, subdirectories, or garbage collection, and files are zero-sized (no read, write, or unlink). But its core functionality is still tricky to implement correctly due to nondeterminism and corner cases like overflows. In fact, the verifier caught two bugs in our initial implementation. The development of YminLFS took one of the authors less than four hours, as detailed next.

2.1 Specification

In Yggdrasil, a file system specification consists of three parts: an abstract data structure representing the logical layout, a set of operations over this data structure to define the intended behavior, and an equivalence predicate that defines whether a given implementation satisfies the specification.

Abstract data structure. We start by specifying the abstract data structure for YminLFS:

```
class FSSpec(BaseSpec):
  def __init__(self):
    self._childmap  = Map((InoT, NameT), InoT)
    self._parentmap = Map(InoT, InoT)
    self._mtimemap  = Map(InoT, U64T)
    self._modemap   = Map(InoT, U64T)
    self._sizemap   = Map(InoT, U64T)
```

The state of the data structure is described by five *abstract maps*, created by calling the `Map` constructor with *abstract types* specifying the map's domain and range. The `childmap` maps a directory inode number and a name to a child inode number; `parentmap` maps an inode number back to its parent directory's inode number; and the remaining maps store inode metadata (mtime, mode, and size). Both `InoT` and `U64T` are 64-bit integer types, and `NameT` is a string type.

The `FSSpec` data structure itself places only weak constraints on the logical layout of YminLFS. For example, it does not rule out layouts in which an inode d contains an inode f according to the `childmap`, but f is not contained in d according to the `parentmap`. The `FSSpec` specification disallows such invalid layouts with a *well-formedness invariant*:

```
def invariant(self):
 ino, name = InoT(), NameT()
 return ForAll([ino, name], Implies(
  self._childmap[(ino, name)] > 0,
  self._parentmap[self._childmap[(ino, name)]] == ino))
```

The invariant says that the parent and child mappings of valid (positive) inode numbers agree with each other. Both `ForAll` and `Implies` are built-in logical operators.

File system operations. Given our logical layout, we can now specify the desired behavior of file system operations. Read-only operations, such as `lookup` and `stat`, are easy to define:

```
def lookup(self, parent, name):
  ino = self._childmap[(parent, name)]
  return ino if ino > 0 else -errno.ENOENT

def stat(self, ino):
  return Stat(size=self._sizemap[ino],
              mode=self._modemap[ino],
              mtime=self._mtimemap[ino])
```

Operations that modify the file system are more complex, as they involve updating the state of the abstract maps. For example, to add a new file to a given directory, `mknod` needs to update all abstract maps as follows:

```
def mknod(self, parent, name, mtime, mode):
  # Name must not exist in parent.
  if self._childmap[(parent, name)] > 0:
    return -errno.EEXIST

  # The new ino must be valid & not already exist.
  ino = InoT()
  assertion(ino > 0)
  assertion(Not(self._parentmap[ino] > 0))

  with self.transaction():
    # Update the directory structure.
    self._childmap[(parent, name)] = ino
    self._parentmap[ino] = parent
    # Initialize inode metadata.
    self._mtimemap[ino] = mtime
    self._modemap[ino]  = mode
    self._sizemap[ino]  = 0

  return ino
```

The `InoT()` constructor returns an abstract inode number, which is constrained to be valid (i.e., positive) and not present in any directory. The changes to the file system are wrapped in a transaction to ensure that they happen atomically or not at all (if the system crashes).

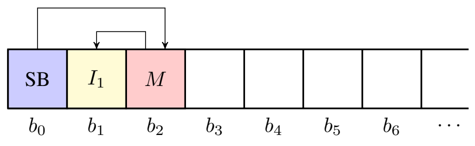
State equivalence predicate. The last part of our YminLFS specification defines what it means for a given file system state to be correct:

```
def equivalence(self, impl):
  ino, name = InoT(), NameT()
  return ForAll([ino, name], And(
    self.lookup(ino, name) == impl.lookup(ino, name),
    Implies(self.lookup(ino, name) > 0,
      self.stat(self.lookup(ino, name)) ==
      impl.stat(impl.lookup(ino, name)))))
```

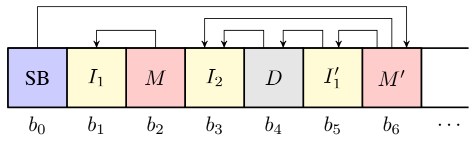
In particular, we require a correct implementation to contain the same files as the abstract data structure, and each file to have the same metadata as its abstract counterpart.

Putting it all together. With our toy specification completed, we now highlight two key features of the Yggdrasil specification approach. First, Yggdrasil specifications are free of implementation details and are therefore reusable. The `FSSpec` data structure does not mandate any particular on-disk layout, nor does it force the implementation to be, for example, a log-structured file system. In fact, our Yxv6 journaling file system is built on top of an extension of this specification (see §4).

Second, Yggdrasil specifications are both succinct and expressive. For example, the specification of `mknod` provides two deep properties in just a few lines of code: crash-free functional correctness (i.e., a file will be created with the correct metadata if there is no crash); and crash safety (i.e., file creation is all-or-nothing even in the face of crashes).



(a) The initial disk state of an empty root directory.



(b) The disk state after adding one file.

Figure 2: YminLFS's on-disk layout. SB is the superblock; I denotes an inode block; M denotes an inode mapping block; D denotes a data block; arrows denote pointers.

2.2 Implementation

To implement a file system in Yggdrasil, the programmer needs to choose a *disk model*, write the code for each specified operation, and write the *consistency invariants* for the on-disk layout. We describe the disk model next, followed by a brief overview of the implementation and consistency invariants for YminLFS. We omit full implementation details (200 lines of Python) for space reasons.

Disk model. Yggdrasil provides several disk models: YminLFS (as well as Yxv6) uses the asynchronous disk model; we will use a synchronous one in §5. The asynchronous disk model specifies a block device that has an unbounded volatile cache and allows arbitrary reordering. Its interface includes the following operations:

- d.`write`(a, v): write a data block v to disk address a;
- d.`read`(a): return a data block at disk address a; and
- d.`flush`$()$: flush the disk cache.

This disk model is trusted to be a correct specification of the underlying physical disk, as we discuss in §4.2. Unless otherwise specified, we assume 64-bit block addresses and 4 KB blocks. We also assume that a single block read/write is atomic, similar to prior work [7, 37].

A log-structured file system. YminLFS is implemented as a log-structured file system that works in a copy-on-write fashion. In particular, it does not overwrite existing blocks (except for the superblock in block zero); it has no garbage collection; and it simply fails when it runs out of blocks, inodes, or directory entries. Its interface provides a `mkfs` operation for initializing the disk, as well as the operations for reading and modifying the file system state that we specified in §2.1.

The `mkfs` operation initializes the disk as shown in Figure 2a. The effect of the operation is to create a file system with a single empty root directory. This involves writing three blocks: the superblock, an inode I_1 for the root directory, and an inode mapping M that stores the mapping from inode numbers to block numbers. After initialization, M has one entry, $1 \mapsto b_1$, and I_1 points to no data blocks, as the root directory is empty. The superblock points to M, and it stores two additional counters: the next available inode number i (which is initialized to 2 since the root is 1) and the next available block number b (which is initialized to 3).

To add a file to the root directory, `mknod` changes the disk state from Figure 2a to Figure 2b, as follows:

1. add an inode block I_2 for the new file;
2. add a data block D for the root directory, which now has one entry that maps the name of the new file to its inode number 2;
3. add an inode block I'_1 for the updated root directory, which points to its data block D;
4. add an inode mapping block M', which has two entries: $1 \mapsto b_5$ and $2 \mapsto b_3$;
5. finally, update the superblock SB to point to the latest inode mapping M'.

Since the disk can reorder these updates, `mknod` must issue disk flushes to be crash-safe. For example, if there is no flush between the last two writes (steps 4 and 5), the disk can reorder them; if the system crashes in between the reordered writes, the superblock will point to garbage data in b_6, resulting in corrupted YminLFS state. For now, we assume a naïve but correct implementation of `mknod` that inserts five flushes, one after each write. In §2.4, we will use the Yggdrasil optimizer to remove the first three flushes.

Consistency invariants. A consistency invariant for a file system implementation is analogous to the well-formedness invariant for its specification—it is a predicate that determines whether a given disk state corresponds to a valid file-system image. Yggdrasil uses consistency invariants for two purposes: push-button verification and run-time checking in the style of `fsck` [20, 30]. For verification, Yggdrasil checks that the invariant holds for the initial file system state right after `mkfs`; in addition, it assumes the consistency invariant as part of the precondition for each operation, and checks that the invariant holds as part of the postcondition. Once the implementation is verified, Yggdrasil can optionally generate an `fsck`-like checker from these invariants (though the checker cannot repair corrupted file systems). Such a checker is useful even for a bug-free file system, as hardware failures and bugs in other parts of the system can damage the file system [40].

The YminLFS consistency invariant constrains three components of the on-disk layout (Figure 2): the superblock *SB*, the inode mapping block M, and the root directory data block D. The superblock constraint requires the next available inode number i to be greater than 1, the next available block number b to be greater than 2, and the pointer to M to be both positive and smaller than b. The inode mapping constraint ensures that M maps each inode number in range $(0, i)$ to a block number in range $(0, b)$. Finally, the root directory constraint requires D to map file names to inode numbers in range $(0, i)$. These three constraints are all Yggdrasil needs to verify YminLFS (see §2.3).

2.3 Verification

To verify that the YminLFS implementation (§2.2) satisfies the `FSSpec` specification (§2.1), Yggdrasil uses the Z3 solver [15] to prove a two-part crash refinement theorem (§3). The first part of the theorem deals with crash-free executions. It requires the implementation and specification to behave alike in the absence of crashes: if both YminLFS and `FSSpec` start in equivalent and consistent states, they end up in equivalent and consistent states. The verifier defines equivalence using the specification's `equivalent` predicate (§2.1), and consistency using the implementation's consistency invariants (§2.2).

The second part of the theorem deals with crashing executions. It requires the implementation to exhibit no more crash states (disk states after a crash) than the specification: each possible state of the YminLFS implementation (including states caused by crashes and reordered writes) must be equivalent to *some* crash state of `FSSpec`.

Counterexamples. If there is any bug in the implementation or consistency invariants, the verifier will generate a *counterexample* to help programmers understand the bug. A counterexample consists of a concrete trace of the implementation that violates the crash refinement theorem. As an example, consider the potential missing flush bug described in §2.2. If we remove the flush between the last two writes in the implementation of `mknod`, Yggdrasil outputs the following counterexample:

```
# Pending writes
lfs.py:167 mknod write(new_imap_blkno, imap)

# Synchronized writes
lfs.py:148 mknod write(new_blkno, new_ino)
lfs.py:154 mknod write(new_parentdata, parentdata)
lfs.py:160 mknod write(new_parentblkno, parentinode)
lfs.py:170 mknod write(SUPERBLOCK, sb)

# Crash point
[..]
lfs.py:171 mknod flush()
```

The output describes the bug by showing the point at which the system crashes and the list of writes pending in the cache (along with their source code locations). In this example, the write of the new inode mapping block (step 4 above) is still pending, but the write to update the superblock to point to that block (step 5) has reached the disk, corrupting YminLFS's state.

The visualization of "pending" and "synchronized" writes in the counterexample is specific to the asynchronous disk model; one can extend Yggdrasil with new disk models and customized visualizations.

Our initial YminLFS implementation contained two other bugs: one in the `lookup` logic and one in the data layout. Neither of the bugs appeared during testing runs. Both bugs were found by the verifier in a matter of seconds, and we quickly localized and fixed them by examining the resulting counterexamples.

Proofs. If the Yggdrasil verifier finds no counterexamples to the crash refinement theorem, then none exist, and we have obtained a *proof* of correctness. In particular, the crash refinement theorem holds for all disks with up to 2^{64} blocks, and for every trace of file system operations, regardless of its length. After we fixed the bugs in our initial YminLFS implementation, the verifier proved its correctness in under 30 seconds.

It is worth noting that the theorem holds if the file system is the only user of the disk. For instance, it does *not* hold if an adversary corrupted the file system image by directly modifying the disk. To address this issue, one can run `fsck` generated by Yggdrasil, which guarantees to detect any such inconsistencies.

2.4 Optimizations and compilation

As described in §2.2, YminLFS's `mknod` implementation uses five disk flushes. Yggdrasil provides a greedy optimizer that tries to remove every disk flush and re-verify the code. Running the optimizer on the `mknod` code removes three out of the five flushes within three minutes, while still guaranteeing correctness.

The optimized and verified YminLFS implementation, which is in Python, is executable but slow. Yggdrasil invokes the Cython compiler [3] to generate C code from Python for better performance. It also provides a small bridge to connect the generated C code to FUSE [17]. The result is a single-threaded user-space file system.

2.5 Summary

We have demonstrated how to specify, implement, debug, verify, optimize, and execute the YminLFS file system using Yggdrasil. Compared to previous file system verification work, push-button verification eases the proof burden and enables automated features such as visualizing bugs and optimizing code.

Since there is no need to manually prove or annotate implementation code when using Yggdrasil, the verification effort is spent mainly on writing the specification and coming up with consistency invariants about the on-disk data format. We find the counterexample visualizer useful for finding bugs in these two parts.

The trusted computing base (TCB) includes the file system specification, Yggdrasil's verifier, visualizer, and compiler (but not the optimizer), their dependencies (i.e., the Z3 solver, Python, and gcc), as well as FUSE and the Linux kernel. See §6 for discussion on limitations.

3 The Yggdrasil architecture

In Yggdrasil, the core notion of correctness is crash refinement. This section gives a formal definition of crash refinement, and describes how Yggdrasil's components use this definition to support verification, counterexample visualization, and optimization.

3.1 Reasoning about systems with crashes

In Yggdrasil, programmers write both specifications and implementations (referred to as "systems" in this section) as state machines: each system comprises a state and a set of operations that transition the state. A transition can occur only if the system is in a consistent state, as determined by its consistency invariant $\mathcal{I}$. This invariant is a predicate over the system's state, indicating whether it is consistent or corrupted; see §2.2 for an example.

Consider a specification F_0 and an implementation F_1. Our goal is to show that F_1 is correct with respect to F_0. Since both systems are state machines, a strawman definition of correctness is that they transition in lock step (i.e., bisimulation): starting from equivalent consistent states, if the same operation is invoked on both systems, they will transition to equivalent consistent states (where equivalence between states is defined by a system-specific predicate). However, this bisimulation-based definition is too strong for systems that interact with external storage, as it does not account for non-determinism from disk reorderings, crashes, or recovery.

To address this shortcoming, we introduce *crash refinement* as a new definition of correctness. At a high level, crash refinement says that F_1 is correct with respect to F_0 if, starting from equivalent consistent states and invoking the same operation on both systems, *any* state produced by F_1 is equivalent to *some* state produced by F_0. To formalize this intuition, we define the behavior of a system in the presence of crashes, formalize crash refinement for individual operations, and extend the resulting definition to entire systems.

System operations. We model the behavior of a system operation with a function f that takes three inputs:

- its current state s;
- an external input $\boldsymbol{x}$, such as data to write; and
- a crash schedule $\boldsymbol{b}$, which is a set of boolean values denoting the occurrence of crash events.

Applying f to these inputs, written as $f(s, \boldsymbol{x}, \boldsymbol{b})$, produces the next state of the system.

As a concrete example, consider a single disk write operation that writes value v to disk address a. The external input to the write operation's function f_w is the pair (a, v). The state s is the disk content before the write; $s(a)$ gives the old value at the address a. The asynchronous disk model in Yggdrasil generates a pair of boolean values $(on, sync)$ as the crash schedule. The on value indicates whether the write operation completed successfully by storing its data into the volatile cache. The $sync$ value indicates whether the write's effect has been synchronized from the volatile cache to stable storage. After executing the write operation, the disk is updated to contain v at the address a only if both on and $sync$ are true, and left unchanged otherwise (e.g., the system crashed before completing the write, or before synchronizing it to stable storage):

$$f_w(s, \boldsymbol{x}, \boldsymbol{b}) = s[a \mapsto \texttt{if } on \land sync \texttt{ then } v \texttt{ else } s(a)],$$
$$\text{where } \boldsymbol{x} = (a, v) \text{ and } \boldsymbol{b} = (on, sync).$$

Crash refinement. To define crash refinement for a given schedule, we start from a special case where write operations always complete and their effects are synchronized to disk. That is, the crash schedule is the constant vector ***true***. Let $s_0 \sim s_1$ denote that s_0 and s_1 are equivalent states according to a user-defined equivalence relation (as in §2.1). We write $s_0 \sim_{\mathcal{I}_0,\mathcal{I}_1} s_1$ to say that s_0 and s_1 are equivalent and consistent according to their respective system invariants $\mathcal{I}_0$ and $\mathcal{I}_1$:

$$s_0 \sim_{\mathcal{I}_0,\mathcal{I}_1} s_1 \triangleq \mathcal{I}_0(s_0) \land \mathcal{I}_1(s_1) \land s_0 \sim s_1.$$

With a crash-free schedule ***true***, two functions f_0 and f_1 are equivalent if they produce equivalent and consistent output states when given the same external input $\boldsymbol{x}$, as well as equivalent and consistent starting states:

Definition 1 (Crash-free equivalence)**.** Given two functions f_0 and f_1 with their system consistency invariants $\mathcal{I}_0$ and $\mathcal{I}_1$, respectively, we say f_0 and f_1 are *crash-free equivalent* if the following holds:

$$\forall s_0, s_1, \boldsymbol{x}.\ (s_0 \sim_{\mathcal{I}_0,\mathcal{I}_1} s_1) \Rightarrow (s_0' \sim_{\mathcal{I}_0,\mathcal{I}_1} s_1')$$
$$\text{where } s_0' = f_0(s_0, \boldsymbol{x}, \boldsymbol{true}) \text{ and } s_1' = f_1(s_1, \boldsymbol{x}, \boldsymbol{true}).$$

Next, we allow for the possibility of crashes. We say that f_1 is correct with respect to f_0 if, for any crash schedule, the state produced by f_1 with that schedule is equivalent to a state produced by f_0 with *some* schedule:

Definition 2 (Crash refinement without recovery)**.** Function f_1 is a *crash refinement (without recovery)* of f_0 if (1) f_0 and f_1 are crash-free equivalent and (2) the following holds:

$$\forall s_0, s_1, \boldsymbol{x}, \boldsymbol{b}_1.\ \exists \boldsymbol{b}_0.\ (s_0 \sim_{\mathcal{I}_0,\mathcal{I}_1} s_1) \Rightarrow (s_0' \sim_{\mathcal{I}_0,\mathcal{I}_1} s_1')$$
$$\text{where } s_0' = f_0(s_0, \boldsymbol{x}, \boldsymbol{b}_0) \text{ and } s_1' = f_1(s_1, \boldsymbol{x}, \boldsymbol{b}_1).$$

Finally, we consider the possibility that the system may run a *recovery function* upon reboot. A recovery function r is a system operation (as defined above) that takes no external input (as it is executed when the system starts). It should also be *idempotent*: even if the system crashes during recovery and re-runs the recovery function many times, the resulting state should be the same once the recovery is complete.

Definition 3 (Recovery idempotence)**.** A recovery function r is idempotent if the following holds:

$$\forall s, \boldsymbol{b}.\ r(s, \boldsymbol{true}) = r(r(s, \boldsymbol{b}), \boldsymbol{true}).$$

Note that this definition accounts for multiple crash-reboot cycles during recovery, by repeated application of the idempotence definition on each intermediate crash state $r(s, \boldsymbol{b})$, $r(r(s, \boldsymbol{b}), \boldsymbol{b'})$, ..., where $\boldsymbol{b}$, $\boldsymbol{b'}$, ... are the schedules for each crash during recovery.

Definition 4 (Crash refinement with recovery)**.** Given two functions f_0 and f_1, their system consistency invariants $\mathcal{I}_0$ and $\mathcal{I}_1$, respectively, and a recovery function r, f_1 with r is a *crash refinement* of f_0 if (1) f_0 and f_1 are crash-free equivalent; (2) r is idempotent; and (3) the following holds:

$$\forall s_0, s_1, \boldsymbol{x}, \boldsymbol{b}_1.\ \exists \boldsymbol{b}_0.\ (s_0 \sim_{\mathcal{I}_0,\mathcal{I}_1} s_1) \Rightarrow (s_0' \sim_{\mathcal{I}_0,\mathcal{I}_1} s_1')$$
$$\text{where } s_0' = f_0(s_0, \boldsymbol{x}, \boldsymbol{b}_0) \text{ and } s_1' = r(f_1(s_1, \boldsymbol{x}, \boldsymbol{b}_1), \boldsymbol{true}).$$

Furthermore, systems may run background operations that do not change the externally visible state of a system (i.e., no-ops), such as garbage collection.

Definition 5 (No-op)**.** Function f with a recovery function r is a *no-op* if (1) r is idempotent, and (2) the following holds:

$$\forall s_0, s_1, \boldsymbol{x}, \boldsymbol{b}_1.\ (s_0 \sim_{\mathcal{I}_0,\mathcal{I}_1} s_1) \Rightarrow (s_0 \sim_{\mathcal{I}_0,\mathcal{I}_1} s_1')$$
$$\text{where } s_1' = r(f(s_1, \boldsymbol{x}, \boldsymbol{b}_1), \boldsymbol{true}).$$

With per-function crash refinement and no-ops, we can now define crash refinement for entire systems.

Definition 6 (System crash refinement)**.** Given two systems F_0 and F_1, and a recovery function r, F_1 is a *crash refinement* of F_0 if every function in F_1 with r is either a crash refinement of the corresponding function in F_0 or a no-op.

The rest of this section will describe Yggdrasil's components based on the definition of crash refinement.

3.2 The verifier

Given two file systems, F_0 and F_1, Yggdrasil's verifier checks that F_1 is a crash refinement of F_0 according to Definition 6. To do so, the verifier performs symbolic execution [6, 24] for each operation $f_i \in F_i$ to obtain an SMT encoding of the operation's output, $f_i(s_i, \boldsymbol{x}, \boldsymbol{b_i})$, when applied to a symbolic input $\boldsymbol{x}$ (represented as a bitvector), symbolic disk state s_i (represented as an uninterpreted function over bitvectors), and symbolic crash schedule $\boldsymbol{b_i}$ (represented as booleans). It then invokes the Z3 solver to check the validity of either the no-op identity (Definition 5) if f_1 is a no-op, or else the per-function crash refinement formula (Definition 4) for the corresponding functions $f_0 \in F_0$ and $f_1 \in F_1$.

To capture *all* execution paths in the SMT encoding of $f_i(s_i, \boldsymbol{x}, \boldsymbol{b_i})$, the verifier adopts a "self-finitizing" symbolic execution scheme [49], which simply unrolls loops and recursion *without* bounding the depth. Since this scheme will fail to terminate on non-finite code, the verifier requires file systems to be implemented in a finite way: for instance, loops must be bounded [50]. In our experience (further discussed in §4), the finiteness requirement does not add much programming burden.

To prove the validity of the per-function crash refinement formula, the verifier uses Z3 to check if the formula's negation is unsatisfiable. If so, the result is a proof that f_1 is a crash refinement of f_0. Otherwise, Z3 produces a *model* of the formula's negation, which represents a concrete counterexample to crash refinement: disk states s_0 and s_1, an input $\boldsymbol{x}$, and a crash schedule $\boldsymbol{b_1}$, such that $s_0 \sim_{\mathcal{I}_0,\mathcal{I}_1} s_1$ but there is no crash schedule $\boldsymbol{b_0}$ that satisfies $f_0(s_0, \boldsymbol{x}, \boldsymbol{b_0}) \sim_{\mathcal{I}_0,\mathcal{I}_1} f_1(s_1, \boldsymbol{x}, \boldsymbol{b_1})$.

Checking the satisfiability of the negated crash refinement formula in Definition 4 requires reasoning about quantifiers. In general, such queries are undecidable. In our case, the problem is decidable because the quantifiers range over finite domains, and the formula is expressed in a decidable combination of decidable theories (i.e., equality with uninterpreted functions and fixed-width bitvectors) [51]. Moreover, Z3 can solve this problem in practice because the crash schedule $\boldsymbol{b_0}$, which is a set of boolean variables, is the only universally quantified variable in the negated formula. As many file system specifications have simple semantics, the crash schedule $\boldsymbol{b_0}$ has few boolean variables—often only one (e.g., the transaction in §2.1)—which makes the reasoning efficient.

The verifier's symbolic execution engine supports all regular Python code with concrete (i.e., non-symbolic) values. For symbolic values, it supports booleans, fixed-width integers, maps, and lists of concrete length, as well as regular control flow including conditionals and loops, but no exceptions or coroutines. It does not support symbolic execution into C library code.

3.3 The counterexample visualizer

To make counterexamples to validity easier to understand, Yggdrasil provides a visualizer for the asynchronous disk model. Given a counterexample model of the formula in Definition 4, the visualizer produces concrete disk event traces (e.g., see §2.3) as follows. First, it uses the crash schedule $\boldsymbol{b_1}$ to identify the boolean variable *on* that indicates where the system crashed, and relates that location to the implementation source code with a stack trace. Second, it evaluates the boolean *sync* variables that indicate whether a write is synchronized to disk, and prints out the pending writes with their corresponding source locations to help identify unintended reorderings. Yggdrasil also allows programmers to supply their own plugin visualizer for data structures specific to their file system images. We found this facility useful when developing YminLFS and Yxv6.

3.4 The optimizer

The Yggdrasil optimizer improves the run-time performance of implementation code. Yggdrasil treats the optimizer as untrusted and re-verifies the optimized code it generates. This simple design, made possible by push-button verification, allows programmers to plug in custom optimizations without the burden of supplying a correctness proof. We provide one built-in optimization that greedily removes disk flush operations (see §2.4), implemented by rewriting the Python abstract syntax tree.

4 The Yxv6 file system

The section describes the design, implementation, and verification of the Yxv6 journaling file system. At a high level, verifying the correctness of Yxv6 requires Yggdrasil to obtain an SMT encoding of both the specification and implementation through symbolic execution, and to invoke an SMT solver to prove the crash refinement theorem. A simple approach, used by YminLFS in §2, is to directly prove crash refinement between the entire file system specification and implementation. However, the complexity of Yxv6 makes such a proof intractable for state-of-the-art SMT solvers. To address this issue, Yxv6 employs a modular design enabled by crash refinement to scale up SMT reasoning.

4.1 Design overview

Yxv6 uses crash refinement to achieve scalable SMT reasoning in three steps. First, to reduce the size of SMT encodings, Yxv6 stacks five layers of abstraction, each consisting of a specification and implementation, starting with an asynchronous disk specification (§4.2). We use Yggdrasil to prove crash refinement theorems for each layer, showing that each correctly implements its specification. Upper layers then use the specifications of lower layers, rather than their implementations, in order to accelerate verification. This layered approach effectively bounds the reasoning to a single layer at a time.

Second, many file system operations touch only a small part of the disk. To allow the SMT solver to exploit this locality, Yxv6 explicitly uses multiple separate disks rather than one. For example, by storing the free bitmap on a separate disk, the SMT solver can easily infer that updating it does not affect the rest of the file system. We then prove crash refinement from this multi-disk system to a more space-efficient file system that uses only a single disk (§4.3). The result of these first two steps is Yxv6+sync, a *synchronous* file system commits a transaction for each system call (by forcing the log to disk), similar to xv6 [14] and FSCQ [7].

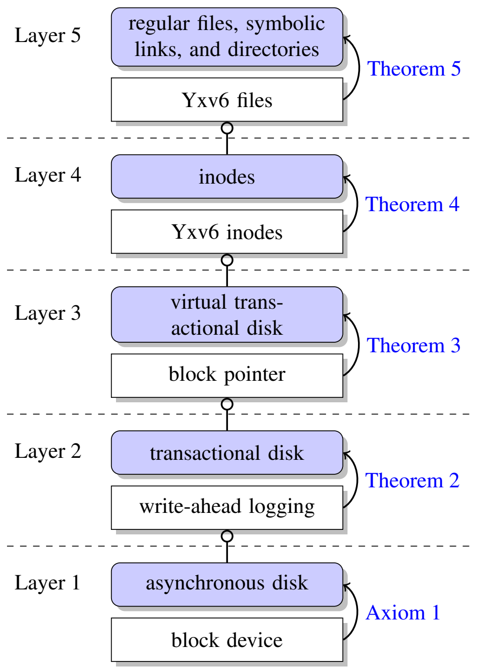


Figure 3: The stack of layers of Yxv6. Within each layer, a shaded box represents the specification; a (white) box represents the implementation; and the implementation is a crash refinement of its specification, denoted using an arrow. Each implementation (except for the lowest layer) builds on top of a specification from the layer below, denoted using a circle.

Finally, for better run-time performance, we implement an optimized variant of Yxv6+sync that groups multiple system calls into one transaction [19] and commits only when the log is full or upon `fsync`. We prove the resulting file system, called Yxv6+group_commit, is a crash refinement of Yxv6+sync with a more relaxed crash consistency model (§4.4).

4.2 Stacking layers of abstraction

Figure 3 shows the five abstraction layers of Yxv6. Each layer consists of a specification and an implementation that is written using a lower-level specification. We describe each of these layers in turn.

Layer 1: Asynchronous disk. The lowest layer of the stack is a specification of an asynchronous disk. This specification comprises the asynchronous disk model we used in §2.2 to implement YminLFS. Since the implementation of a physical block device is opaque, we assume the specification correctly models the block device (i.e., the specification is more conservative and allows more behavior than real hardware), as follows:

Axiom 1. A block device is a crash refinement of the asynchronous disk specification.

Layer 2: Transactional disk. The next layer introduces the abstraction of a transactional disk, which manages multiple separate data disks, and offers the following operations:

- d.`begin_tx`() starts a transaction;
- d.`commit_tx`() commits a transaction;
- d.`write_tx`(j, a, v) adds to the current transaction a write of value v to address a on disk j; and
- d.`read`(j, a) returns the value at address a on disk j.

The specification says that operations executed within the same transaction are atomic (i.e., all-or-nothing) and sequential (i.e., transactions cannot be reordered).

The implementation uses the standard write-ahead logging technique [19, 31]. It uses one asynchronous disk (from layer 1) for the log, and a set of asynchronous disks for data. Using a single transactional disk to manage multiple data disks allows higher layers to separate writes within a transaction (e.g., updates to data and inode blocks will not interfere), which helps scale SMT reasoning; §4.3 refines the multiple disks to one.

The implementation is parameterized by the transaction size limit k (i.e., the maximum number of writes in one transaction). The log disk uses a fixed number of blocks, determined by k, as a header to store log entry addresses, and the remaining blocks to store log entry data. The first entry in the first header block is a counter of log entries; the consistency invariant for the transactional disk layer says that this counter is always zero after recovery. The Yxv6+sync file system sets $k = 10$, while Yxv6+group_commit sets $k = 511$. For each of these settings, we prove the following theorem:

Theorem 2. The write-ahead logging implementation is a crash refinement of the transactional disk specification.

Layer 3: Virtual transactional disk. The specification of the virtual transactional disk is similar to that of the transactional disk, but instead uses 64-bit *virtual disk addresses* [22]. Each virtual address can be mapped to a physical disk address or unmapped later; reads and writes are valid for mapped addresses only. We will use this abstraction to implement inodes in the upper layer.

The virtual transactional disk implementation uses the standard block pointers approach. It uses one transactional disk managing at least three data disks: one to store the free block bitmap, another to store direct block pointers, and the third to store both data and singly indirect block pointers (higher layers will add additional disks). The free block bitmap disk stores only one bit in each of its blocks, which simplifies SMT reasoning but wastes disk space; §4.3 will refine it to a more space-efficient version.

The implementation relies on two consistency invariants: (1) the mapping from virtual disk addresses to physical disk addresses is injective (i.e., each physical address is mapped at most once), and (2) if a virtual disk address is mapped to physical address a, the a^{th} bit in the block bitmap must be marked as used. We use these invariants to prove the following theorem:

Theorem 3. The block pointer implementation is a crash refinement of the virtual transactional disk specification.

Layer 4: Inodes. The fourth layer introduces the abstraction of inodes. Each inode is uniquely identified using a 32-bit inode number. The specification maps an inode number to 2^{32} blocks, and to a set of metadata such as `size`, `mtime`, and `mode`.

The implementation is straightforward thanks to the virtual transactional disk specification. It simply splits the 64-bit virtual disk address space into 2^{32} ranges, and each inode takes one range, which has 2^{32} "virtual" blocks, similar to NVMFS/DFS [22]. Inode metadata resides on a separate disk managed by the virtual transactional disk (which now has four data disks). There are no consistency invariants in this layer. We prove the following theorem:

Theorem 4. The Yxv6 inode implementation is a crash refinement of the inode specification.

Layer 5: File system. The top layer of the file system is an extended version of `FSSpec` given in §2, with regular files, directories, and symbolic links.

The implementation builds on top of the inode specification, using a separate inode bitmap disk and another for orphan inodes. Both are managed by the virtual transactional disk (which now has six data disks plus the log disk, giving a total of seven disks). There are two consistency invariants: (1) if an inode is not marked as used in the inode bitmap disk, its size must be zero in the metadata; and (2) if an inode has n blocks, no "virtual" block larger than n is mapped. Using these invariants, we prove the final crash refinement theorem:

Theorem 5. The Yxv6 implementation of files is a crash refinement of the specification of regular files, symbolic links, and directories.

Finitization. The Yggdrasil verifier requires Yxv6 operations to be finite, as mentioned in §3.2. Most file system operations satisfy this requirement, as they use only a small number of disk reads and writes. For example, moving a file involves updating only the source and destination directories. However, there are two exceptions.

First, search-related procedures, such as finding a free bit in a bitmap, may need to read many blocks. We choose *not* to verify the bit-finding algorithm, but instead adopt the idea of validation [38, 46, 48] to implement such search algorithms. The validator, which we do verify, simply checks that an index returned by the search is indeed marked free in the bitmap and if not, fails the operation with an error code. We use similar

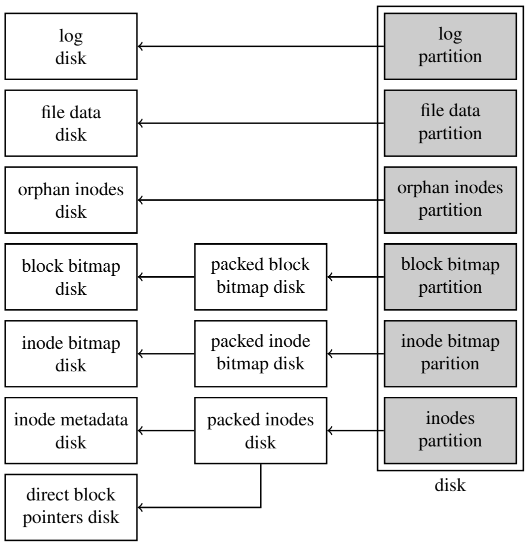


Figure 4: The refinement of disk layout of the Yxv6 file system, from multiple disks to a single disk. The arrows $A \leftarrow B$ denote that B is a crash refinement of A.

strategies for directory entry lookup. This approach allows us to treat search procedures as a black box, absolving the SMT solver from the need to reason about the many paths through the algorithm.

The second case is unlinking a file, as freeing all its data blocks needs to write potentially many blocks. To finitize this operation, our implementation simply moves the inode of the file into a special orphan inodes disk, which is a finite operation, and relies on a separate garbage collector to reclaim the data blocks at a later time. We further prove that reclamation is a no-op (as per the definition in §3.1), as freeing a block referenced by the orphan inodes disk does not affect the externally visible state of the file system. We will summarize the trade-offs of validation in §4.5.

4.3 Refining disk layouts

Theorem 5 gives a file system that runs on seven disks: the write-ahead log, the file data, the block and inode bitmaps for managing free space, the inode metadata, the direct block pointers, and the orphan inodes. Using separate disks scales SMT reasoning, but it has two downsides. First, the two bitmaps use only one bit per block and the inode metadata disk stores one inode per block, wasting space. Second, requiring seven disks makes the file system difficult to use. We now prove with crash refinement that it is correct to pack these disks into one disk (Figure 4) similar to the xv6 file system [14].

Intuitively, it is correct to pack multiple blocks that store data sparsely into one with a dense representation, because the packed disk has the same or fewer possible disk states. For instance, bitmap disks used in §4.2 store one bit per block; the n-th bit of the bitmap is stored in the lowest bit of block n. On the other hand, a *packed* bitmap disk stores $4\,\text{KB} \times 8 = 2^{15}$ bits per block, and the n-th bit is stored in bit $n \bmod 2^{15}$ of block $n/2^{15}$. Clearly, using the packed bitmap is a crash refinement of the sparse one. The same holds for using packed inodes. Similarly, a single disk with multiple non-overlapping partitions exhibits fewer states than multiple disks; for example, a flush on a single disk will flush all the partitions, but not for multiple disks. Combining these packing steps, we prove the following theorem:

Theorem 6. The Yxv6 implementation using seven non-overlapping partitions of one asynchronous disk, with packed bitmaps and inodes, is a crash refinement of that using seven asynchronous disks.

4.4 Refining crash consistency models

Theorem 6 gives a *synchronous* file system that commits a transaction for each system call. This file system, which we call Yxv6+sync, incurs a slowdown as it flushes the disk frequently (see §8 for performance evaluation). The Yxv6+group_commit file system implements a more relaxed crash consistency model [5, 37]. Unlike Yxv6+sync, its write-ahead logging implementation groups multiple transactions together [19].

Intuitively, doing a single combined transaction produces fewer possible disk states compared to two separate transactions, as in the latter scheme the system can crash in between the two and expose the intermediate state. We prove the following theorem:

Theorem 7. Yxv6+group_commit is a crash refinement of Yxv6+sync.

4.5 Summary of design trade-offs

Unlike conventional journaling file systems, the first Yxv6 design in §4.2 uses multiple disks. To decide the number of disks, we adopt a simple guideline: whenever a part of the disk is *logically* separate from the rest of the file system, such as the log or the free bitmap, we assign a separate disk for that part. In our experience, this is effective in scaling up SMT reasoning.

Yxv6's final on-disk layout closely resembles that of the xv6 and FSCQ file systems. One notable difference is that Yxv6 uses an orphan inodes partition to manage files that are still open but have been unlinked, similarly to the orphan inode list [21] in ext3 and ext4. This design ensures correct atomicity behavior of `unlink` and `rename`, especially when running with FUSE, which xv6 and FSCQ do not guarantee.

Another difference to FSCQ is that Yxv6 uses validation instead of verification in managing free blocks and inodes. Although the resulting allocator is safe, it does not guarantee that block or inode allocation will succeed when there is enough space, treating such failures as a quality-of-service issue.

5 Beyond file systems

Although we designed Yggdrasil for writing verified file systems, the idea of crash refinement generalizes to applications that use disks in other ways. This section describes two examples: Ycp, a file copy utility; and Ylog, a persistent log data structure.

The Ycp file copy utility. Like the Unix `cp` utility, Ycp copies the contents of one file to another. Unlike `cp`, it has a formal specification: if the copy operation succeeds, the file system is updated so that the target file contains the same data as the source file; if it fails due to a system crash or an invalid target (e.g., a directory or a symbolic link), the file system is unchanged.

The implementation of Ycp uses the Yxv6 file system specification (Figure 3). It follows a common atomicity pattern: (1) create a temporary file, (2) write the source data to it, and (3) rename it to atomically create the target file. There is no consistency invariant as Ycp uses file system operations and is independent of disk layout.

We verify that the implementation of Ycp is a crash refinement of its specification using Yggdrasil. This shows that Yggdrasil and Yxv6's specification are useful for reasoning about application-level correctness.

The Ylog persistent log. Ylog is a verified implementation of the persistent log from the Arrakis operating system [36]. The Arrakis log is designed to provide an efficient storage API with strong atomicity and persistence guarantees. The core logging operation is a multi-block append, which extends an on-disk log with entries that can span multiple blocks. This append operation must appear to be both atomic and immediately persistent, even in the presence of crashes.

The Arrakis persistent log was originally designed to run on top of an LSI Logic MegaRAID SAS-3 3108 RAID controller with a battery-backed cache. We therefore chose to implement Ylog on top of a *synchronous* disk model, which does not reorder writes and matches the behavior of the RAID controller. Ylog uses the same on-disk layout as Arrakis: the first block (i.e., superblock) contains metadata, such as the number of entries and a pointer to the end of the log, followed by blocks that contain the data of each entry.

When comparing Ylog's implementation with that of Arrakis, we discovered two bugs in the Arrakis persistent log: its crash recovery logic was *not* idempotent, and the log could end up with garbage data if the system crashed again during recovery. The bugs were reported to and confirmed by the Arrakis developers.

6 Discussion

This section discusses the limitations of Yggdrasil, as well as our experience using and designing the toolkit.

component	specification	implementation	consistency inv
Yxv6	250	1,500	5
YminLFS	25	150	5
Ycp	15	45	0
Ylog	35	60	0
infrastructure	–	1,500	–
FUSE stub	–	250	–

Figure 5: Lines of code for the Yggdrasil toolkit and storage systems built using it, excluding blank lines and comments.

Limitations. Yggdrasil reasons about single-threaded code, so file systems written using Yggdrasil do not support concurrency. Cython [3], Yggdrasil's Python-to-C compiler, is unverified, although we have not yet encountered any bugs in the development.

Yggdrasil relies on SMT solvers for automated reasoning, and is limited to first-order logic. It is less expressive than interactive theorem provers such as Coq or Isabelle, although our experience shows that it is sufficient for writing and verifying file systems like Yxv6 based on crash refinement.

Since the Z3 solver is at the core of Yggdrasil, its correctness is critical. To understand this risk, we ran the Yxv6 verification using every buildable snapshot of the Z3 Git repository over the past three years, a total of 1,417 versions. We also used two other SMT solvers, Boolector [32] and MathSAT 5 [9], for cross-checking. We did not observe any inconsistent results.

The Yxv6 file system lacks several modern file system features, such as extents and delayed allocation in ext4. Compared to hand-written file system checkers, its `fsck` tool is generated by Yggdrasil and guaranteed to detect any violations of consistency invariants, but it cannot repair corrupted file systems.

Lessons learned. Bitvector operations and reasoning about non-determinism (e.g., crashes) are common in file system implementations. These characteristics motivated us to formulate file system verification as an SMT problem, exploiting the fully automated decision procedures for the theories of bitvectors and uninterpreted functions. In addition, using SMT enables Yggdrasil to produce and visualize counterexamples; we find this ability useful for tracking subtle file system bugs during development, especially corner cases such as overflows and missing flushes [27].

In earlier development of Yggdrasil, we struggled to find a disk representation for scalable SMT reasoning. We explored several approaches, such as a lazy list of symbolic blocks (e.g., EXE [53]) and the theory of arrays, all resulting in a verification bottleneck.

Yggdrasil represents a disk using uninterpreted functions that map a block address and an in-block offset to a 64-bit integer. This two-level map helped to scale up verification. Mapping to 64-bit integers also allowed

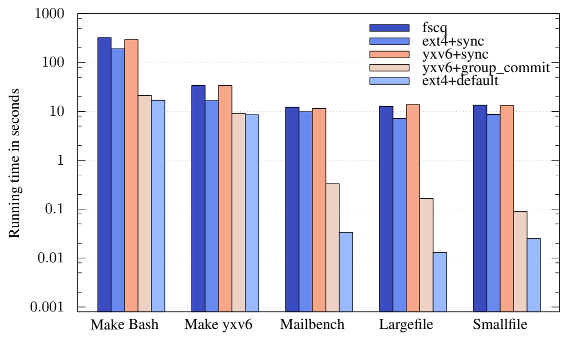


Figure 6: Performance of file systems on an SSD, in seconds (log scale; lower is better).

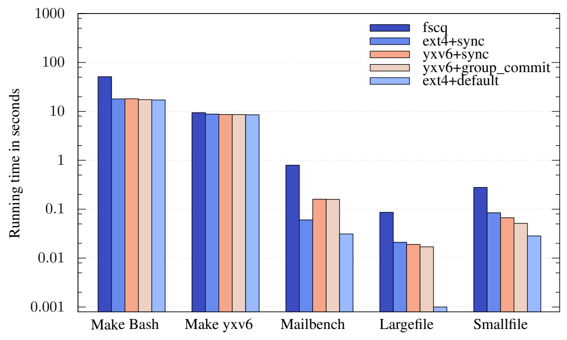


Figure 7: Performance of file systems on a RAM disk, in seconds (log scale; lower is better).

Yggdrasil to generate efficient C code. The idea of separating logical and physical data representations using crash refinement further reduced the verification time by orders of magnitude. As we will show in §8, verifying Yxv6+sync's theorems took less than a minute, thanks to Z3's efficient decision procedures, whereas Coq took 11 hours to check the proofs of FSCQ [7] (which has similar features to Yxv6+sync).

Crash refinement requires programmers to design a system as a state machine and implement each operation in a finite way. File systems fit well into this paradigm. We have used crash refinement in several contexts: to stack layers of abstraction, to pack multiple blocks or disks, and to relax crash consistency models. Crash refinement does not require advanced knowledge of program logics (e.g., separation logic [41] in FSCQ), and is amenable to automated SMT reasoning.

7 Implementation

Figure 5 lists the code size of the file systems and other storage applications built using Yggdrasil, the common infrastructure code, and the FUSE boilerplate. In total, they consist of about 4,000 lines of Python code.

8 Evaluation

This section uses Yxv6 as a representative example to evaluate file systems built using Yggdrasil. We aim to answer the following questions:

- Does Yxv6 provide end-to-end correctness?
- What is the run-time performance?
- What is the verification performance?

Unless otherwise noted, all experiments were conducted on a 4.0 GHz quad-core Intel i7-4790K CPU running Linux 4.4.0.

Correctness. We tested the correctness of Yxv6 as follows. First, we ran it on existing benchmarks. Both Yxv6+sync and Yxv6+group_commit passed the `fsstress` tests from the Linux Test Project [26]; they also passed the SibylFS POSIX conformance tests [42], except for incomplete features such as hard links or extended attributes. Second, we have been using Yxv6 to self-host Yggdrasil's development since early March, including the writing of this paper; our experience is that it is reliable for daily use. Third, we applied the disk block enumerator from the Ferrite toolkit [5] (similar to the Block Order Breaker [37]) to cross-check that the file system state was consistent after a crash and recovery.

To test the correctness of Yxv6's `fsck`, we manually corrupted file system images by overwriting them with random bytes; Yxv6's `fsck` was able to detect corruption in all these cases.

Run-time performance. To understand the run-time performance of Yxv6, we ran a set of five benchmarks similar to those used in FSCQ [7]: compiling the source code of `bash` and Yxv6, running a mail server from the sv6 operating system [10], and the LFS benchmark [44].

We compare the two Yxv6 variants against the verified file system FSCQ and the ext4 file system in two configurations: its default configuration (i.e., `data=ordered`), and with `data=journal+sync` options, which together are similar to Yxv6+sync. Although Yxv6's implementation is closest to xv6, we excluded xv6's performance numbers as it crashed frequently on three benchmarks and did not pass the `fsstress` tests.

Figure 6 shows the on-disk performance with all the file systems running on a Samsung 850 PRO SSD. The y-axis shows total running time in seconds (log scale). We see that Yxv6+sync performs similarly to FSCQ and to ext4's slower configuration. Yxv6+group_commit, which groups several operations into a single transaction, outperforms those file systems by 3–150× and is on average within 10× of ext4's default configuration.

To understand the CPU overhead, we repeated the experiments using a RAM disk, as shown in Figure 7. The two variants of Yxv6 have similar performance numbers. They both outperform FSCQ, and are close in performance to ext4 (except for the `largefile` benchmark). We believe the reason is that Yxv6 benefits from Yggdrasil's Python-to-C compiler, while FSCQ's performance is affected by its use of Haskell code extracted from Coq.

Verification performance. As we mentioned in §6, the total verification time for Yxv6+sync is under a minute on a single core. It achieved this verification performance due to Z3's efficient SMT solving and the use of crash refinement in the file system.

Verifying Yxv6+group_commit took a longer time, because it is parameterized to use larger transactions (see §4.2). It finished within 1.6 hours using 24 cores (Intel Xeon 2.2 GHz), approximately 36 hours on a single core.

9 Related work

Verified file system implementations. Developers looking to build and verify file systems have primarily turned to interactive theorem provers such as Coq [11] and Isabelle [33]. Our approach is most similar to FSCQ [7], a verified crash-safe file system developed in Coq. Their proof shows that after reboot, FSCQ's recovery routines will correctly recover the file system state without data loss. These theorems are stated in *crash Hoare logic*, which extends Hoare logic with support for crash conditions and recovery procedures. Our approach also bears similarities to Flashix [16, 47], another verified crash-safe file system. The Flashix proof consists of several refinements from the POSIX specification layer down to an implementation which can be extracted to Scala. These refinements are proved in the KIV interactive theorem prover in terms of abstract state machines.

Compared to these examples, Yggdrasil's push-button verification substantially lowers the proof burden. Yggdrasil can verify the Yxv6 implementation given only the specifications and five consistency invariants. This ease of verification, together with richer debugging support, also helped us implement several optimizations in Yxv6 that make its performance 3–150× faster than FSCQ and within 10× of ext4.

COGENT [2] takes a different approach to building verified file systems, defining a new restricted language together with a certified compiler to C code. The COGENT language rules out several common sources of errors, such as memory safety and memory leaks, reducing the verification proof burden. We believe Yggdrasil and COGENT to be complimentary: on one hand, COGENT provides certified extraction to C code which could replace Yggdrasil's unverified extraction from Python; on the other hand, Yggdrasil's crash refinement strategy could help COGENT to produce more automated proofs.

File system specifications and crash consistency. Several projects have developed formal specifications of file systems. SibylFS [42] is an effort to formalize POSIX interfaces and test implementation conformance. But because POSIX file system interfaces underspecify allowed crash behavior, so does the SibylFS formalization. Commuter [10] formalizes the commutativity of POSIX interface calls to study scalability, but as with SibylFS, the formalization does not consider crashes.

Modern file systems adopt various crash recovery strategies, including write-ahead logging (or journaling) [19, 31], log-structured file systems [44], copy-on-write (or shadowing) [4, 43], and soft updates [18, 29]. This diversity complicates reasoning about application-level crash safety. Pillai et al. [37] and Zheng et al. [55] surveyed the crash safety of real-world applications, finding many crash-safety bugs despite extensive engineering effort to tolerate and recover from crashes. Bornholt et al. [5] formalized the crash guarantees of modern file systems as *crash-consistency models*, to help application writers provide crash safety. A formally verified file system can provide these models as an artifact of the verification process. Yggdrasil's crash refinement strategy helps to abstract low-level implementation details out of these application-facing models.

Bug-finding tools. Rather than building a new verified file system, several existing projects focus on finding bugs in existing file systems. FiSC [54] and eXplode [52] use model checking to find consistency bugs. ELEVEN82 [25] is a bug-finding tool for "recoverability bugs," where a system can crash in such a way that even after recovery, the file system is left in a state not reachable by any crash-free execution. Yggdrasil is complementary to these tools: ELEVEN82's automata-based bug detection allows it to explore complex optimizations, while Yggdrasil provides proofs not only of crash safety but of functional correctness.

10 Conclusion

Yggdrasil presents a new approach for building file systems with the aid of push-button verification. It guarantees correctness through a definition of file system crash refinement that is amenable to efficient SMT solving. It introduces several techniques to scale up automated verification, including the stack of abstractions and the separation of data representations. We believe that this is a promising direction since it provides a strong correctness guarantee with a low proof burden. All of Yggdrasil's source code is publicly available at `http://locore.cs.washington.edu/yggdrasil/`.

Acknowledgments

We thank Helga Gudmundsdottir, Niel Lebeck, Hank Levy, Haohui Mai, Qiao Zhang, the anonymous reviewers, and our shepherd, Petros Maniatis, for their feedback. This work was supported in part by DARPA under contract FA8750-16-2-0032 and by a gift from the VMware University Research Fund.

References

[1] R. Alagappan, V. Chidambaram, T. S. Pillai, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. Beyond storage APIs: Provable semantics for storage stacks. In *Proceedings of the 15th Workshop on Hot Topics in Operating Systems (HotOS)*, Kartause Ittingen, Switzerland, May 2015.

[2] S. Amani, A. Hixon, Z. Chen, C. Rizkallah, P. Chubb, L. O'Connor, J. Beeren, Y. Nagashima, J. Lim, T. Sewell, J. Tuong, G. Keller, T. Murray, G. Klein, and G. Heiser. COGENT: Verifying high-assurance file system implementations. In *Proceedings of the 21st International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 175–188, Atlanta, GA, Apr. 2016.

[3] S. Behnel, R. Bradshaw, C. Citro, L. Dalcin, D. Seljebotn, and K. Smith. Cython: The best of both worlds. *Computing in Science Engineering*, 13(2): 31–39, Mar.–Apr. 2011. http://cython.org/.

[4] J. Bonwick. ZFS: The last word in filesystems, Oct. 2005. https://blogs.oracle.com/bonwick/entry/zfs_the_last_word_in.

[5] J. Bornholt, A. Kaufmann, J. Li, A. Krishnamurthy, E. Torlak, and X. Wang. Specifying and checking file system crash-consistency models. In *Proceedings of the 21st International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 83–98, Atlanta, GA, Apr. 2016.

[6] C. Cadar, D. Dunbar, and D. Engler. KLEE: Unassisted and automatic generation of high-coverage tests for complex systems programs. In *Proceedings of the 8th Symposium on Operating Systems Design and Implementation (OSDI)*, pages 209–224, San Diego, CA, Dec. 2008.

[7] H. Chen, D. Ziegler, T. Chajed, A. Chlipala, M. F. Kaashoek, and N. Zeldovich. Using Crash Hoare Logic for certifying the FSCQ file system. In *Proceedings of the 25th ACM Symposium on Operating Systems Principles (SOSP)*, Monterey, CA, Oct. 2015.

[8] H. Chen, D. Ziegler, A. Chlipala, M. F. Kaashoek, E. Kohler, and N. Zeldovich. Specifying crash safety for storage systems. In *Proceedings of the 15th Workshop on Hot Topics in Operating Systems (HotOS)*, Kartause Ittingen, Switzerland, May 2015.

[9] A. Cimatti, A. Griggio, B. Schaafsma, and R. Sebastiani. The MathSAT5 SMT solver. In *Proceedings of the 19th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 93–107, Rome, Italy, Mar. 2013.

[10] A. T. Clements, M. F. Kaashoek, N. Zeldovich, R. T. Morris, and E. Kohler. The scalable commutativity rule: Designing scalable software for multicore processors. In *Proceedings of the 24th ACM Symposium on Operating Systems Principles (SOSP)*, pages 1–17, Farmington, PA, Nov. 2013.

[11] Coq development team. *The Coq Proof Assistant Reference Manual, Version 8.5pl2*. INRIA, July 2016. http://coq.inria.fr/distrib/current/refman/.

[12] J. Corbet. Thoughts on the ext4 panic, Oct. 2012. https://lwn.net/Articles/521803/.

[13] J. Corbet. A tale of two data-corruption bugs, May 2015. https://lwn.net/Articles/645720/.

[14] R. Cox, M. F. Kaashoek, and R. T. Morris. Xv6, a simple Unix-like teaching operating system, 2016. http://pdos.csail.mit.edu/6.828/xv6.

[15] L. de Moura and N. Bjørner. Z3: An efficient SMT solver. In *Proceedings of the 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 337–340, Budapest, Hungary, Mar.–Apr. 2008.

[16] G. Ernst, J. Pfähler, G. Schellhorn, and W. Reif. Inside a verified flash file system: Transactions & garbage collection. In *Proceedings of the 7th Working Conference on Verified Software: Theories, Tools and Experiments*, San Francisco, CA, July 2015.

[17] FUSE. Filesystem in userspace, 2016. https://github.com/libfuse/libfuse.

[18] G. R. Ganger and Y. N. Patt. Metadata update performance in file systems. In *Proceedings of the 1st Symposium on Operating Systems Design and Implementation (OSDI)*, pages 49–60, Monterey, CA, Nov. 1994.

[19] R. Hagmann. Reimplementing the Cedar file system using logging and group commit. In *Proceedings of the 11th ACM Symposium on Operating Systems Principles (SOSP)*, pages 155–162, Austin, TX, Nov. 1987.

[20] V. Henson. The many faces of fsck, Sept. 2007. https://lwn.net/Articles/248180/.

[21] V. Henson, Z. Brown, T. Ts'o, and A. van de Ven. Reducing fsck time for ext2 file systems. In *Proceedings of the Linux Symposium*, pages 395–408, Ottawa, Canada, June 2006.

[22] W. K. Josephson, L. A. Bongo, D. Flynn, and K. Li. DFS: A file system for virtualized flash storage. In *Proceedings of the 8th USENIX Conference on File and Storage Technologies (FAST)*, pages 1–15, San Jose, CA, Feb. 2010.

[23] R. Joshi and G. J. Holzmann. A mini challenge: Build a verifiable filesystem. *Formal Aspects of Computing*, 19(2):269–272, June 2007.

[24] J. C. King. Symbolic execution and program testing. *Communications of the ACM*, 19(7):385–394, July 1976.

[25] E. Koskinen and J. Yang. Reducing crash recoverability to reachability. In *Proceedings of the 43rd ACM Symposium on Principles of Programming Languages (POPL)*, pages 97–108, St. Petersburg, FL, Jan. 2016.

[26] LTP. Linux Test Project, 2016. http://linux-test-project.github.io/.

[27] L. Lu, A. C. Arpaci-Dusseau, R. H. Arpaci-Dusseau, and S. Lu. A study of Linux file system evolution. *ACM Transactions on Storage*, 10(1): 31–44, Jan. 2014.

[28] W. M. McKeeman. Peephole optimization. *Communications of the ACM*, 8:443–444, July 1965.

[29] M. K. McKusick. Journaled soft-updates. In *BSDCan*, Ottawa, Canada, May 2010.

[30] M. K. McKusick and T. J. Kowalski. Fsck—the UNIX file system check program. In *UNIX System Manager's Manual (SMM), 4.4 Berkeley Software Distribution*. University of California, Berkeley, Oct. 1996.

[31] C. Mohan, D. Haderle, B. Lindsay, H. Pirahesh, and P. Schwarz. ARIES: A transaction recovery method supporting fine-granularity locking and partial rollbacks using write-ahead logging. *ACM Transactions on Database Systems*, 17(1):94–162, Mar. 1992.

[32] A. Niemetz, M. Preiner, and A. Biere. Boolector 2.0. *Journal on Satisfiability, Boolean Modeling and Computation (JSAT)*, 9:53–58, 2015.

[33] T. Nipkow, L. C. Paulson, and M. Wenzel. *Isabelle/HOL: A Proof Assistant for Higher-Order Logic*. Springer-Verlag, Feb. 2016.

[34] L. O'Connor, Z. Chen, C. Rizkallah, S. Amani, J. Lim, T. Murray, Y. Nagashima, T. Sewell, and G. Klein. Refinement through restraint: Bringing down the cost of verification. In *Proceedings of the 21st ACM SIGPLAN International Conference on Functional Programming (ICFP)*, pages 89–102, Nara, Japan, Sept. 2016.

[35] N. Palix, G. Thomas, S. Saha, C. Calvès, J. L. Lawall, and G. Muller. Faults in Linux: Ten years later. In *Proceedings of the 16th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 305–318, Newport Beach, CA, Mar. 2011.

[36] S. Peter, J. Li, I. Zhang, D. R. K. Ports, D. Woos, A. Krishnamurthy, T. Anderson, and T. Roscoe. Arrakis: The operating system is the control plane. In *Proceedings of the 11th Symposium on Operating Systems Design and Implementation (OSDI)*, pages 1–16, Broomfield, CO, Oct. 2014.

[37] T. S. Pillai, V. Chidambaram, R. Alagappan, S. Al-Kiswany, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. All file systems are not created equal: On the complexity of crafting crash-consistent applications. In *Proceedings of the 11th Symposium on Operating Systems Design and Implementation (OSDI)*, pages 433–448, Broomfield, CO, Oct. 2014.

[38] A. Pnueli, M. Siegel, and E. Singerman. Translation validation. In *Proceedings of the 4th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 151–166, Lisbon, Portugal, Mar.–Apr. 1998.

[39] V. Prabhakaran, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. Model-based failure analysis of journaling file systems. In *Proceedings of the 35th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, pages 802–811, Yokohama, Japan, June–July 2005.

[40] V. Prabhakaran, L. N. Bairavasundaram, N. Agrawal, H. S. Gunawi, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. IRON file systems. In *Proceedings of the 20th ACM Symposium on Operating Systems Principles (SOSP)*, pages 206–220, Brighton, UK, Oct. 2005.

[41] J. C. Reynolds. Separation logic: A logic for shared mutable data structures. In *Proceedings of the 17th Annual IEEE Symposium on Logic in Computer Science*, pages 55–74, Copenhagen, Denmark, July 2002.

[42] T. Ridge, D. Sheets, T. Tuerk, A. Giugliano, A. Madhavapeddy, and P. Sewell. SibylFS: formal specification and oracle-based testing for POSIX and real-world file systems. In *Proceedings of the 25th ACM Symposium on Operating Systems Principles (SOSP)*, pages 38–53, Monterey, CA, Oct. 2015.

[43] O. Rodeh, J. Bacik, and C. Mason. BTRFS: The Linux B-tree filesystem. *ACM Transactions on Storage*, 9(3), Aug. 2013.

[44] M. Rosenblum and J. Ousterhout. The design and implementation of a log-structured file system. In *Proceedings of the 13th ACM Symposium on Operating Systems Principles (SOSP)*, pages 1–15, Pacific Grove, CA, Oct. 1991.

[45] C. Rubio-González, H. S. Gunawi, B. Liblit, R. H. Arpaci-Dusseau, and A. C. Arpaci-Dusseau. Error propagation analysis for file systems. In *Proceedings of the 2009 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 270–280, Dublin, Ireland, June 2009.

[46] H. Samet. Proving the correctness of heuristically optimized code. *Communications of the ACM*, 21 (7):570–582, July 1978.

[47] G. Schellhorn, G. Ernst, J. Pfähler, D. Haneberg, and W. Reif. Development of a verified flash file system. In *Proceedings of the ABZ Conference*, June 2014.

[48] T. Sewell, M. Myreen, and G. Klein. Translation validation for a verified OS kernel. In *Proceedings of the 2013 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 471–482, Seattle, WA, June 2013.

[49] E. Torlak and R. Bodik. A lightweight symbolic virtual machine for solver-aided host languages. In *Proceedings of the 2014 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 530–541, Edinburgh, UK, June 2014.

[50] L. Torvalds. Re: [patch] measurements, numbers about `CONFIG_OPTIMIZE_INLINING=y` impact, Jan. 2009. `https://lkml.org/lkml/2009/1/9/497`.

[51] C. M. Wintersteiger, Y. Hamadi, and L. de Moura. Efficiently solving quantified bit-vector formulas. In *Proceedings of the 10th Conference on Formal Methods in Computer-Aided Design*, pages 239–246, Lugano, Switzerland, Oct. 2010.

[52] J. Yang, P. Twohey, D. Engler, and M. Musuvathi. Using model checking to find serious file system errors. In *Proceedings of the 6th Symposium on Operating Systems Design and Implementation (OSDI)*, pages 273–287, San Francisco, CA, Dec. 2004.

[53] J. Yang, C. Sar, P. Twohey, C. Cadar, and D. Engler. Automatically generating malicious disks using symbolic execution. In *Proceedings of the 27th IEEE Symposium on Security and Privacy*, pages 243–257, Oakland, CA, May 2006.

[54] J. Yang, P. Twohey, D. Engler, and M. Musuvathi. eXplode: A lightweight, general system for finding serious storage system errors. In *Proceedings of the 7th Symposium on Operating Systems Design and Implementation (OSDI)*, pages 131–146, Seattle, WA, Nov. 2006.

[55] M. Zheng, J. Tucek, D. Huang, F. Qin, M. Lillibridge, E. S. Yang, B. W. Zhao, and S. Singh. Torturing databases for fun and profit. In *Proceedings of the 11th Symposium on Operating Systems Design and Implementation (OSDI)*, pages 449–464, Broomfield, CO, Oct. 2014.

Intermittent Computation without Hardware Support or Programmer Intervention

Joel Van Der Woude
Sandia National Laboratories[*]

Matthew Hicks
University of Michigan

Abstract

As computation scales downward in area, the limitations imposed by the batteries required to power that computation become more pronounced. Thus, many future devices will forgo batteries and harvest energy from their environment. Harvested energy, with its frequent power cycles, is at odds with current models of long-running computation.

To enable the correct execution of long-running applications on harvested energy—without requiring special-purpose hardware or programmer intervention—we propose Ratchet. Ratchet is a compiler that adds lightweight checkpoints to unmodified programs that allow existing programs to execute across power cycles correctly. Ratchet leverages the idea of idempotency, decomposing programs into a continuous stream of re-executable sections connected by lightweight checkpoints, stored in non-volatile memory. We implement Ratchet on top of LLVM, targeted at embedded systems with high-performance non-volatile main memory. Using eight embedded systems benchmarks, we show that Ratchet correctly stretches program execution across frequent, random power cycles. Experimental results show that Ratchet enables a range of existing programs to run on intermittent power, with total run-time overhead averaging below 60%—comparable to approaches that require hardware support or programmer intervention.

1 Introduction

Improvements in the design and development of computing hardware have driven hardware size and cost to rapidly shrink as performance improves. While early computers took up entire rooms, emerging computers are millimeter-scale devices with the hopes of widespread deployment for sensor network applications [23]. These rapid changes drive us closer to the realization of smart dust [20], enabling applications where the cost and size of computation had previously been prohibitive. We are rapidly approaching a world where computers are not just your laptop or smart phone, but are integral parts your clothing [47], home [9], or even groceries [4].

[*]Work completed while at the University of Michigan

Unfortunately, while the smaller size and lower cost of microcontrollers enables new applications, their ubiquitous adoption is limited by the form factor and expense of batteries. Batteries take up an increasing amount of space and weight in an embedded system and require special thought to placement in order to facilitate replacing batteries when they die [20]. In addition, while Moore's Law drives the development of more powerful computers, batteries have not kept pace with the scaling of computation [18]. Embedded systems designers attempt to address this growing gap by leveraging increasingly power-efficient processors and design practices [49]. Unfortunately, these advances have hit a wall—the battery wall; enabling a dramatic change in computing necessitates moving to batteryless devices.

Batteryless devices, instead of getting energy from the power grid or a battery, harvest their energy from their environment (*e.g.,* sunlight or radio waves). In fact, the first wave of energy harvesting devices is available today [4, 50, 9]. These first generation devices prove that it is possible to compute on harvested energy. This affords system designers the novel opportunity to remove a major cost and limitation to system scaling.

Unfortunately, while harvested energy represents an opportunity for system designers, it represents a challenge for software developers. The challenge comes from the nature of harvested energy: energy harvesting provides insufficient power to perform long-running, continuous, computation [37]. This results in frequent power losses, forcing a program to restart from the be-

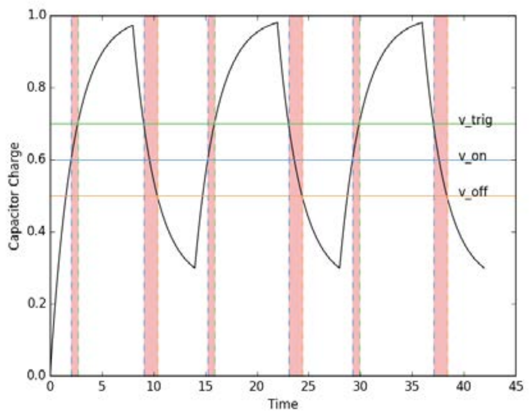


Figure 1: Energy harvesting devices replace batteries with a small energy storage capacitor. This is the ideal charge and decay of that capacitor when given a square wave input power source. The voltage across the capacitor must be above `v_trig` for program computation to take place as that is the minimum energy required to store the *largest* possible checkpoint in the *worst-case* environmental and device conditions. The energy expended going from `v_on` to `v_trig` and from `v_trig` to `v_off` is wasted in what is called the guard band. The guard band is essential for correctness in one-time checkpointing systems [39, 17, 3]. In real systems, the charge and discharge rate is chaotic, depending on many variables, including temperature and device orientation.

ginning, in hopes of more abundant power next time. While leveraging faster non-volatile memory technologies might seem like an easy way to avoid the problems associated with these frequent power cycles, previous work exposes the inconsistent states that can result from power cycles when using these technologies [26, 38].

Previous research attempts to address the problem of intermittent computation using two broad approaches: rely on specialized hardware [39, 29, 3, 17, 27] or require the programmer to reason about the effects of common case power failures on mixed-volatility systems [26]. Hardware-based solutions, rely on a single checkpoint that gets saved just before power runs out. It is critical for correctness that every bit of work gets saved by the checkpoint and that there is no (non-checkpointed) work after a checkpoint [38]. On the other hand, taking a checkpoint too early wastes energy after the checkpoint that could be used to perform meaningful work. This tradeoff mandates specialized hardware to measure available power and predict when power is likely to fail. Due to the intermittent nature of harvested energy, predicting power failure is a risky venture that requires large guard bands to accommodate a range of environmental conditions and hardware variances. Figure 1 depicts how guard-bands waste energy that could otherwise be used to make forward progress. In addition to the guard-bands wasting energy, the power monitoring hardware itself consumes power. Even simple power monitoring circuits (think 1-bit voltage level detector) consume power up to 33% of the power of modern ultra-low-power microcontrollers [5, 43].

An alternative approach, as taken by DINO [26], is to forgo specialized hardware, instead, placing the burden on the programmer to reason about the possible outcomes of frequent, random, power failures. DINO requires that programmers divide programs into a series of checkpoint-connected tasks. These tasks then use data versioning to ensure that power cycles do not violate memory consistency. Smaller tasks increase the likelihood of eventual completion, at the cost of increased overhead. Larger tasks result in fewer checkpoints, but risk never completing execution. Thus, the burden is on the programmer to implement—for all control flows—correct-sized tasks given the program and the expected operating environment. Note that even small changes in the program or operating environment can change dramatically the optimal task structure for a program.

Our goal is to answer the question: *What can be done without requiring hardware modifications or burdening the programmer?* To answer this question, we propose leveraging information available to the compiler to preserve memory consistency without input from the programmer or specialized hardware. We draw upon the wealth of research in fault tolerance [31, 24, 21, 51, 25, 13] and static analysis [7, 22] and construct Ratchet, a compiler that is able to decompose unmodified programs into a series of re-executable sections, as shown in Figure 2. Using static analysis, the compiler can separate code into idempotent sections—*i.e.,* sequences of code that can be re-executed without entering a state inconsistent with program semantics. The compiler identifies idempotent sections by looking for loads and stores to non-volatile memory and then enforcing that no section contains a write after read (WAR) to the same address. By decomposing programs down to a series of re-executable sections and gluing them together with checkpoints of volatile state, Ratchet supports existing, arbitrary-length programs, no matter the power source. Ratchet shifts the burden of reasoning about the effects of intermittent computation and mixed-volatility away from the programmer to the compiler—without relying on hardware support.

We implement Ratchet as a set of modifications to the LLVM compiler [22], targeting energy harvesting platforms that use the ARM architecture [2] and have wholly non-volatile main memory. We also implement an ARM-based energy-harvesting simulator that simulates power failures at frequencies experienced by existing energy harvesting devices. To benchmark Ratchet, we use it to instrument the newlib C-library [45], libgcc, and eight embedded system benchmarks [14]. Finally, we verify the correctness of Ratchet by executing all instrumented benchmarks, over a range of power cycle rates, on our simulator, which includes a set of memory consistency invariants dynamically check for idempotence violations. Our experimental results show that Ratchet correctly stretches program execution across frequent, random power cycles, while adding 60% total run-time overhead[1]. This is comparable to approaches that require hardware support or programmer intervention and much better than the alternative of most benchmarks never completing execution with harvested energy.

This paper makes several key contributions:

- We design and implement the first software-only system that *automatically* and *correctly* stretches program execution across random, frequent power cycles. As a part of our design, we extend the notion of idempotency to memory.
- We evaluate Ratchet on a wider range of benchmarks than previously explored and show that its total run-time overhead is competitive with existing solutions that require hardware support or programmer intervention.
- We open source Ratchet, including our energy harvesting simulator [16], our benchmarks [15], and modifications to LLVM to implement Ratchet [44].

2 Background

The emergence of energy harvesting devices poses the question: *How can we perform long-running computation with unreliable power?* Answering this question forces us to go beyond a direct application of existing work from the fault tolerance community due to four properties of energy harvesting devices:

- Power availability and duration are unknowable for most use cases.
- Added energy drain by hardware is just as important as added cycles by software.
- Small variations in the device and the environment have large effects on up time.
- Faults (*i.e.,* power cycles) are the common case.

[1]We say total run-time overhead to cover all sources of run-time overhead, including hardware and software. Keep in mind that adding hardware indirectly increases the run time of programs by decreasing the amount of energy available for executing instructions. Because Ratchet is software only, the total run-time overhead is equal to the overhead due to saving checkpoints and re-execution.

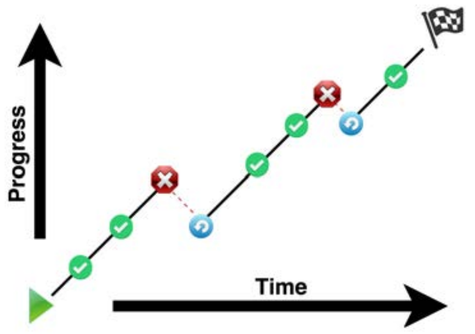


Figure 2: Ratchet-compiled program in operation. The checkmarks represent completed checkpoints, the x's represent power failures, and the dashed lines to the backward rotating arrows represent the system restarting execution at the latest checkpoint. This figure shows how programs execute as normal with added overhead from checkpoints and re-execution.

These four properties dictate how system builders construct energy harvesting devices and how researchers make programs amenable to intermittent computation.

2.1 Prediction vs. Resilience

Given the properties of harvested energy, there are two checkpointing methods for enabling long-running computation: one-time and continuous. One-time checkpointing approaches attempt to predict when energy is about to run out and checkpoint all volatile state right before it does. Doing this requires measuring the voltage across the energy storage capacitor as depicted in Figure 1. Measuring the voltage requires an Analog-to-Digital Converter (ADC) configured to measure the capacitor's voltage. Hibernus [3], the lowest overhead one-time checkpointing approach, utilizes an advanced ADC with interrupt functionality and a configurable voltage threshold that removes the need to periodically check the voltage from software. As Table 1 shows, this produces very low overheads, with the two main sources of overhead being the extra power consumed by the ADC and the energy wasted waiting in the guard bands.

While many energy-harvesting systems have ADCs, the program may require use of the ADC, the ADC may not support interrupts, or the ADC may not be configured (in hardware) to monitor the voltage of the energy storage capacitor. Without such an ADC, programs must be able to fail at any time and still complete execution correctly. Making programs resilient to spontaneous power failures is the domain of continuous checkpointing systems. Continuous checkpointing systems must maintain the abstraction of *re-execution memory consistency* (*i.e.,* a section of code is unable to determine if it is being executed for the first time or being re-executed by exam-

Property	Commodity Checkpointing [32, 11, 31, 34]	Energy Harvesting HW-assisted [17, 3, 29]	DINO [26]	Idempotence [7, 13]	Ratchet
Failure Rate	Days/Weeks	100 ms	100 ms	Days/Weeks	100 ms
Requires	Varies	HW+Compiler	Programmer+Compiler	Compiler	Compiler
Failure Type	Transient Fault	Power Loss	Power Loss	Transient Fault	Power Loss
Memory type	DRAM+HDD	FRAM	SRAM+FRAM	DRAM+HDD	FRAM
Chkpt. Trigger	Time	Low Voltage	Task Boundary	Register WAR	NV WAR
Chkpt. Contents	Varies	VS	VS+NV_TV	—	VS
Overhead	Varies	0–145% [3]	80–170%	0–30%	0–115%
Primary Factor	Varies	Measurement	Task Size	# Faults	Section Size

Table 1: Requirements and behavior of different checkpointing/logging techniques. `WAR` represents a Write-After-Read dependence, `VS` represents Volatile State (*e.g.,* SRAM and registers), `NV` represents Non-Volatile memory (*e.g.,* FRAM), `NV_TV` represents Task Variables stored in Non-Volatile memory, and `Measurement` represents the added time and energy consumed by using voltage-monitoring hardware.

ining memory)[2]. To maintain re-execution memory consistency, continuous checkpointing systems periodically checkpoint volatile state and guard against inconsistent updates to non-volatile memory. DINO [26] does this through data versioning, while Ratchet does this through maintaining idempotence. Table 1 shows that this class of approach also yields low total overheads, with the primary source being time spent checkpointing.

2.2 Memory Volatility

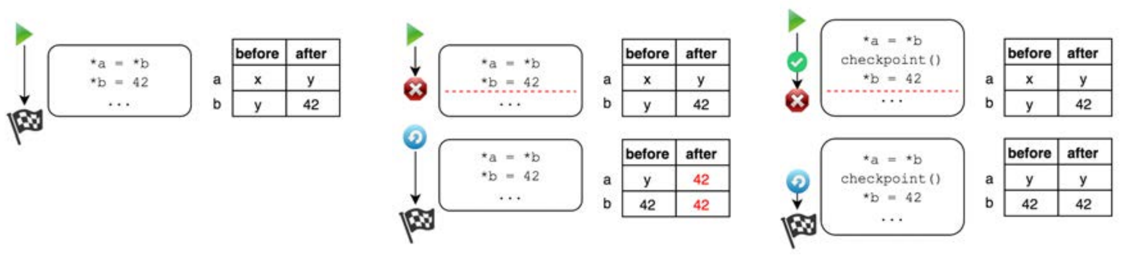
Another consideration for energy harvesting devices is the type, placement, and use of non-volatile memory. While the initial exploration into support for energy harvesting devices, Mementos [39], focuses on supporting Flash-based systems with mixed-volatility main memory, all known follow-on work focuses on emerging systems with Ferroelectric RAM (FRAM)-based, non-volatile, main memory [17, 3, 26, 27]. This transition is necessary as several properties of Flash make it antithetical to harvested energy. The primary reason Flash is ill suited is its energy requirements. Flash works by pushing charge across a dielectric. Doing so is an energy intense operation requiring high voltage that makes little sense when the system is about to run out of power. In fact, on MSP430 devices, Flash writes fail at a much higher voltage than the processor itself fails [5]—increasing the energy wasted in the guard band. A second limitation of Flash is that most programs avoid placing variables there, increasing the amount of volatile state that requires checkpointing. Flash writes, beyond being energy expensive, are slow and complex. Updating a variable stored in Flash requires erasing a much larger block of memory and rewriting all data, along with the one updated value. This process adds complexity to applications and increases write latency over FRAM by two-orders of magnitude.

[2]Note that this is a relaxation on the requirement for deterministic re-execution [48, 31, 30, 33], where it is required that each re-execution produce the same exact result. Our problem only requires that re-executions produce a semantically correct execution.

In comparison with Flash memory, FRAM boasts extremely low voltage writes, as low as a single volt [35]. Writes to FRAM are also nearly as fast as writes to SRAM and are bit-wise programmable. The flexibility and low overheads of FRAM allows for processor designers to create wholly non-volatile main memory, decreasing the size and cost of checkpoints. This opens the door to continuous checkpointing systems as the cost of checkpointing is outweighed by the power requirement of the ADC. While, like previous approaches, we focus on FRAM due to its commercial availability, there are competing non-volatile memory technologies (*e.g.,* Magnetoresistive RAM [46] and Phase Change RAM [40]) that we expect to work equally well with Ratchet. Moving program data to non-volatile memory does come with a cost: previous work reveals that mixing non-volatile and volatile memory is prone to error [38, 26]. Ratchet deals with this by pushing such complexity into the compiler, relieving the programmer from the burden of reasoning about mixed volatility main memory and the effects of power cycles.

3 Design

Ratchet seeks to extend computation across common case power cycles in order to enable programs on energy harvesting devices to complete long-running computation with unreliable power sources. Ratchet enables re-execution after a power failure by saving volatile state to non-volatile memory during compiler-inserted checkpoints. However, checkpointing alone is insuffi-



(a) **(b)** **(c)**

Figure 3: The same code executed without failures, with failures, and with failures with Ratchet. This basic example illustrates one of the difficulties with using non-volatile main memory on intermittently powered computers: failures can create states not possible given program semantics.

cient to ensure correct computation due to the problems with maintaining consistency between volatile and non-volatile memory give unpredictable failures [38, 26]. To ensure correct re-execution, we use compiler analysis to determine sections of code that may be re-executed from the beginning, without producing different results; a property called idempotence. After decomposing programs into idempotent sections, we connect the independent sections with checkpoints of volatile state. This ensures that after a power failure the program will resume with a view of all memory identical to the first (attempted) execution[3].

3.1 Idempotent Sections

Idempotent sections are useful because they are naturally side effect free. Nothing needs to be changed about them in order to protect against memory consistency errors that may arise from partial execution. By recognizing code sections with this property, Ratchet is able to find potentially long sequences of instructions that can be re-executed without any additional work required to ensure memory consistency.

De Kruijf *et al.* identify idempotent sections by looking for instruction sequences that perform a Write-After-Read (WAR) to the same memory address [7]. Under normal execution, overwriting a previously read memory location is inconsequential. However, on systems that roll back to a previous state for recovery (due to potential issues such as power failures), overwriting a value that was previously read will cause a different value to be read when the code section is re-executed. Figure 3 shows an example of how re-executing a section of code with a WAR dependency may introduce execution that diverges from program semantics.

In order to prevent these potential consistency problems Ratchet inserts a checkpoint between the write and the read. This breaks the dependency, separating the read and the write in different idempotent sections. This ensures that the read always gets the original value and the write will be contained in a different idempotent section, where it is free to update the value. Note that a sequence of instructions that contains a WAR may still be idempotent if there exists a write to the same memory address before the first read. For example a WARAW dependency chain is idempotent since the first write initializes the value stored at the memory address so that even if it is changed by the last write, it will be restored upon re-execution before the address is read again. Note that this holds for a potentially infinite sequence of writes and reads, the sequence will be idempotent if there is a write before the first read.

It is important to remember that in order for a load followed by a store cause consistency problems, they must read and modify the same memory. In order to determine which instructions rely on the same memory locations, Ratchet uses intraprocedural alias analysis due to its availability, performance, and precision. Alias analysis *conservatively* identifies instructions that *may* read or modify the same memory locations. Since the alias analysis is intraprocedural we conservatively assume all stores to addresses outside of the stack frame may alias with loads that occurred in the caller. This forces Ratchet to insert a checkpoint along any control flow path that includes a store to non-local memory.

After finding all WARs we use a modified hitting set algorithm to insert the minimum number of checkpoints between the loads and stores. The algorithm works by assigning weights to different points along the control flow graph based upon metrics such as loop depth and the number of other idempotency violations intersected. It uses these metrics to identify checkpointing locations that prevent all possible idempotency violations while trying to avoid locations that will be re-executed more

[3]We are not saying the memory must be identical. We are saying that the values that a given idempotent section of code reads are identical to the initial execution. Idempotency enables this relaxation.

often than necessary. For example, do not checkpoint within a loop if a checkpoint outside the loop will separate all WARs. For a more in-depth discussion of this algorithm, we refer you to de Kruijf *et al.* [7].

3.2 Implicit Idempotency Violations

While looking for WARs identifies the majority of code sequences that violate idempotence, some instructions may implicitly violate idempotence. A pop instruction can be modeled by a read and subsequent update to the stack pointer. This update immediately invalidates the memory locations just read by allowing future push instructions to overwrite the old values. On a system with interrupts, this scenario occurs when an interrupt fires after a pop instruction. In this case, the pop instruction will read from the stack and update the stack pointer. When the interrupt occurs it will perform a push instruction to callee save registers, in order to preserve the state of the processor before the interrupt fired. However, the state saved by the interrupt is written to the stack addresses that were read by the initial pop. If the system re-executes the pop instruction, it will read a value from the interrupt handler—not the original value! This behavior forces Ratchet to treat all pop instructions as implicit idempotency violations since interrupts are not predictable at compile time.

In order to enable a checkpoint before the slots on the stack are freed Ratchet exchanges pop instructions for a series of instructions that perform the same function. The first duty of the pop instruction is to retrieve a set of values from the stack and insert them into registers. Ratchet emulates this by inserting a series of load instructions to retrieve these values from the stack and place them in their respective registers. Note that the load instructions do not update the stack pointer so any interrupt that fires will push the new values above these values. After the data is retrieved from the stack, Ratchet inserts a checkpoint. Finally, Ratchet inserts an update to the stack pointer to free the space previously occupied by the values that we just loaded into registers. By emitting this sequence of instructions we have deconstructed an atomic read write instruction that is an implicit idempotency violation and replaced it by a series of instructions that enable separation of potential idempotency violations.

3.3 Checkpoints

In between each naturally occurring idempotent section, we insert checkpoints in order to save all volatile memory necessary to restart from the failure. In emerging systems we observe non-volatile memory moving closer and closer to the CPU, so far that it has non-volatile RAM [42]. In such a system, all that is needed to restore state are the values stored in the registers. In fact, not all registers are necessary, only registers that are used as inputs to instructions that occur after the checkpoint location. These registers are denoted live-in registers.

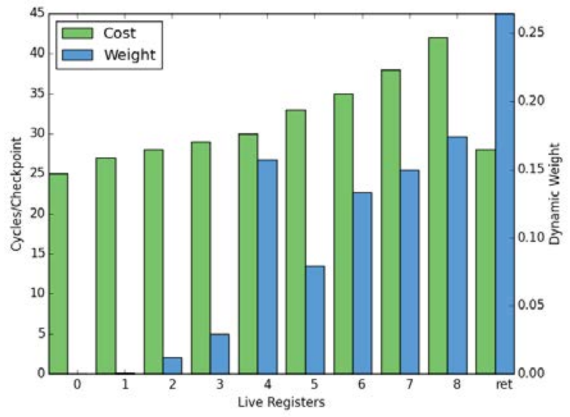


Figure 4: Shows the relationship between checkpoint overhead and live registers. A checkpoint's cost is the number of cycles it takes to commit and its weight refers to how often they occur in our benchmarks relative to the total number of checkpoints.

Traditionally, compilers keep a list of live-in and live-out registers to determine which registers are needed in a basic block to perform some computation and which ones are unused and can be reallocated to reduce register spilling. This information is available to the compiler after registers have been allocated. We are interested in which registers are live-in to a checkpoint because they denote the volatile memory needed to correctly restart from a given location in a program. Figure 4 shows the relationship between checkpoint overhead and number of live-in registers.

In order to prevent power failures during a checkpoint from causing an inconsistent state, we use a double buffering scheme. One buffer holds the previous checkpoint, while the other is used for writing a new checkpoint. A checkpoint is committed by updating the pointer to the valid checkpoint buffer as the last part of writing the checkpoint. We tolerate failures even while taking a checkpoint by never overwriting a checkpoint until a more recent checkpoint is available in the other buffer. The atomicity of the store instruction for a singe word ensures that we always have a valid checkpoint regardless of when we experience a power failure.

3.4 Recovery

In order to recover from a power failure, we insert code before the main function to check to see if there exists a valid checkpoint. If so, we determine that we have experienced a power failure and need to restore state, otherwise we begin executing from main. Restoring state consists of moving all saved registers into the appropriate physical registers. Once the program counter has been restored, execution will restart from the instruction after the most recently executed checkpoint.

In the case that some idempotent sections are too long, that is, power may repeatedly fail before a checkpoint is reached, we use a timer that triggers an interrupt in order to ensure forward progress. Each interrupt checks to see if a new checkpoint has been taken since the last time it was called. It does this by zeroing-out the program counter value in the unset checkpoint buffer each time it is called. It can then tell if a checkpoint has been taken since its last call and only checkpoint if the program counter of the unused checkpoint buffer is still zero.

When checkpointing from the timer interrupt it is impossible to foresee which registers are still live and we must instead conservatively save all of the registers to non-volatile memory. There exists a trade off when selecting the timer speed. A timer that is short increases the overhead due to checkpointing, while a timer that is long increases re-execution overhead. Without additional hardware to measure environmental conditions, the timer can be set on the order of experts estimation of average lifetime for transiently powered devices [39]. Note that for our benchmarks, a timer was not needed in order to make forward progress.

3.5 Challenges

During implementation of Ratchet we encountered a number of design challenges with actually implementing our ideal design. Most of these challenges related to being entrenched at different levels of abstraction within the compiler. While the code is in the compiler's intermediate representation (IR), powerful alias analysis and freedom from architecture level specifics made for a logical location to identify WAR dependencies and insert checkpoints. However, these decisions are dependent on the choices made during the translation from the compiler's IR to machine code, such as which calls are tail calls and information about when register pressure causes register spilling.

This semantic gap causes conservative decision making about where to place checkpoints, since the decisions made in the front end rely on assumptions about where checkpoints will be inserted in the back-end.

3.6 Optimizations

As we began to implement our design, it became clear that we were inserting checkpoints more frequently than needed. As a result of profiling our initial design we implemented several optimizations to remove redundant checkpoints.

```
r1 = mem[sp + 4]        r1 = mem[sp + 4]
checkpoint()            r3 = mem[sp + 8]
mem[sp + 4] = r2        checkpoint()
...                     mem[sp + 4] = r2
r3 = mem[r2 + 8]        ...
checkpoint()            checkpoint()
mem[r2 + 8] = r4        mem[sp + 8] = r4
        a.                      b.
```

Figure 5: Two possible code sequences. In (a) each WAR dependency is separated by a checkpoint, but the two checkpoints cannot be combined without violating idempotency. In (b), the second checkpoint could be moved to the same line as the first checkpoint since there are no potentially aliasing reads separating the two checkpoints.

3.6.1 Interprocedural Idempotency Violations

Because of the limits on interprocedural alias analysis, we initially conservatively inserted a checkpoint on function entry. This protects against potential WAR violations between caller and callee code. We observed that this was often conservative and could be relaxed. A checkpoint is only necessary on function entry if there exists a write that may alias with an address outside of the function's local stack frame. In the presence of an offending write, the function entry is modeled as the potentially offending read (since an offending load could have occurred in the caller).

In addition, we noticed that some tail calls could be implemented without any checkpoints. Since tail calls operate on the stack frame of their caller, a checkpoint on return is unnecessary, assuming that the tail call does not modify non-local memory. However, opportunities for this optimization were observed to be limited due to the difficulty of determining where to put checkpoints for intraprocedural WAR dependencies. This is a result of the semantic gap between compiler stages, when identifying WAR dependencies, we do not yet have perfect information about which calls can be represented as tail calls. We imagine a more extensive version of Ratchet that takes information from each stage of the compiler pipeline and iteratively adjusts checkpoint locations to find the near-minimal set of checkpoints that maintain correctness.

3.6.2 Redundant Checkpoints

Due to the semantic gap between our alias analysis and insertion of checkpoints in the front-end of the compiler (while the code is still in IR), and the instruction scheduling of the back-end, we observed cases where optimizations or other scheduling decisions caused redundant checkpoints. We consider redundant checkpoints to be a pair of checkpoints where any potential idempotency violations could be protected with a single checkpoint. In general, a checkpoint can be relocated within its basic block to any dominating location does not cross any reads and any dominated location that does not cross any writes. This conservative rule follows even without knowing alias information, which allows us to reorder instructions after machine code has been generated and we no longer know which instructions generated the WAR dependency. Figure 5 shows an example of how checkpoints can be combined safely by relocation.

3.6.3 Special Purpose Registers

Since all volatile state must be saved during a checkpoint, all live special purpose registers must be saved along with the live general-purpose registers. Some of the special purpose registers have higher costs to save than the general purpose registers. In our experience implementing Ratchet, we found the cost of checkpointing condition codes to be high. Instead of paying this overhead, we ensured checkpoints were placed after the condition code information was consumed, while still ensuring all non-volatile memory idempotency violations were cut. Ratchet does this by reordering instructions to ensure a checkpoint is not placed between a condition code generator and all possible consumers. This instruction reordering is done with the same constraints as combining checkpoints.

3.7 Architecture Specific Tradeoffs

There are a number of architectural decisions that influence the overhead of our design. Register-rich architectures reduce the number of idempotent section breaks required by reducing the frequency of register spills, at the cost of increasing checkpoint size. Atomic read-modify-write instructions are incompatible with our design since there is no way to checkpoint between the read and the write. On such an architecture, Ratchet could separate the instruction into separate load and store operations by our compiler implementation.

4 Implementation

We implement Ratchet using the LLVM compiler infrastructure [22]. Beyond verifying that Ratchet output executes correctly on an ARM development board [41], we build an ARMv6-M [2] energy-harvesting simulator, with wholly non-volatile memory. The simulator allows for fine-grain control over the frequency, arrival time, and effects of power cycles, as well as allowing us to verify Ratchet's correctness. The benchmarks we use for evaluation all depend on libc, so we also use Ratchet to instrument newlib [45], and link our benchmarks against our instrumented library.

4.1 Compiler

We build our compiler implementation on top of the LLVM infrastructure [22]. We add a front end, IR level pass that detects idempotent section breaks by tracking loads and stores to non-volatile memory that may alias (based on earlier work targeted at registers [7]). This top-level pass inserts checkpoint placeholders that are eventually passed to the back end where machine code is eventually emitted. The back end replaces pop instructions with non-destructive reads and a checkpoint followed by an update to the stack pointer. Next, the back end relocates the inserted placeholders to minimize the number of checkpoints required by combining them and avoiding bisecting condition code def-use chains. After register allocation, each placeholder is replaced by a function call to the checkpointing routine that saves the fewest possible registers. We determine the minimal set of registers to save using the liveness information available in the back end.

One compiler-inserted idempotency-violating construct that we modify the compiler to prevent is the use of shared stack slots for virtual registers. If the compiler were able to re-assign the same stack slot to a different virtual register, it would create an idempotency violation as the original virtual register value is overwritten by a different virtual register's value. While it is possible to include some backend analysis to uncover such situations, we choose to sacrifice some stack space and prevent the sharing of stack slots. Achieving this in LLVM is as simple as adding the flag `-no-stack-slot-sharing` to the compile command. In addition, we include the `mem2reg` optimization that causes some variables that would otherwise be stored in memory to be stored in registers. This reduces the number of idempotency violations thereby reducing the number of checkpoints and increasing idempotent section length.

4.2 Energy Harvesting Simulator

We also implement a cycle accurate ARMv6-M simulator. Many of the coming internet of things class devices are choosing to use ARM devices as they are the performance per Watt leader. As the price of new non-volatile memory technologies decreases and their speed increases, we expect ARM to follow in the footsteps of the MSP430 [42] and move to a wholly non-volatile main memory. In fact, even Texas Instruments, the maker of the MSP430, recently moved to ARM for their newest MSP devices [43].

An energy-harvesting simulator is required because of the difficulties associated with developing and debugging intermittently powered devices. Using a cycle accurate

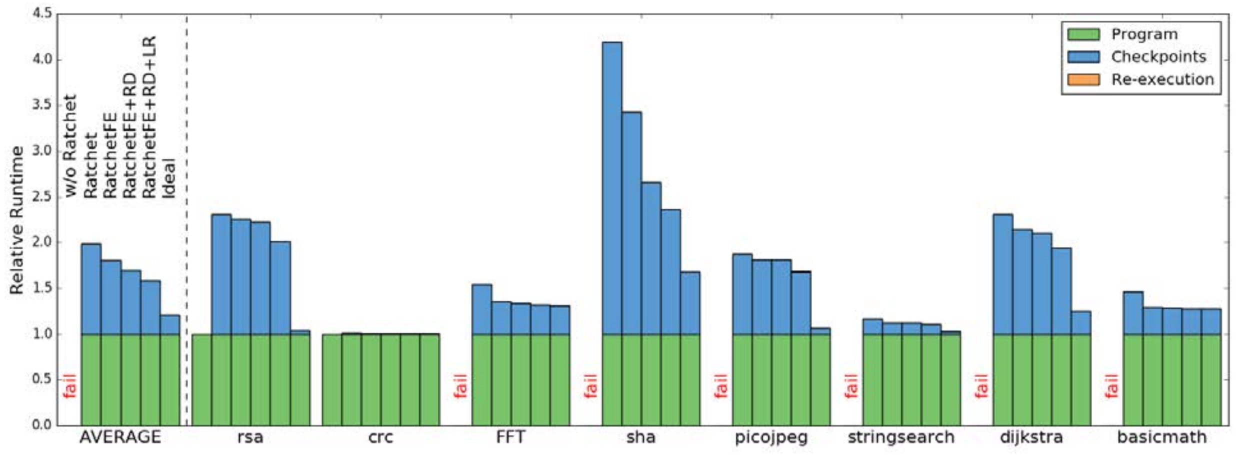


Figure 6: Runtime overhead for several versions of Ratchet.

simulator we are able to simulate failures with a probability distribution that can model the true frequency and effects of power failures experienced by devices in the real world. Using a simulator also allows us to take precise measurements of how much progress a program makes per power cycle and the cycles consumed by re-execution. Lastly, our simulator allows for us to verify the correctness of Ratchet for every benchmark run.

4.3 Idempotent Libraries

In order to ensure that each section of code that runs is idempotent, we instrument all of the libraries needed by the device. In order to facilitate real applications we compile newlib [45], a basic libc and libm library aimed at embedded systems, with Ratchet. This requires a modifying three lines in newlib's makefile to prevent it from building these optimized versions of libc calls. We did this because any uninstrumented code could cause memory consistency to be violated (due to idempotency violations) if power fails after a write-after-read dependency and before a checkpoint.

Lastly, we produce an instrumented version of the minimum runtime functions expected by clang that are included in the compiler-rt project [1]. These functions implement the libgcc interfaces expected by most compilers. As with newlib, we use only the bare minimum optimized assembly implementations. Those that we use, we insert checkpoints by hand between potential write-after-read dependencies. Thankfully, this only needs to be done once by the library's author.

Note that Ratchet supports assembly as long as the assembly is free of idempotence violations or all potential idempotency violations are separated by checkpoints. With additional engineering effort, it is possible to create a tool that inserts these checkpoints automatically through static analysis of the assembly [8].

5 Evaluation

In order to provide a comparison against other checkpointing solutions for energy harvesting devices, we evaluate Ratchet on benchmarks common to these approaches, namely, RSA, CRC, and FFT. For a more complete analysis, we port[4] several benchmarks from MiBench, a set of embedded systems benchmarks categorized by run-time behavior [14]. Expanding the benchmark set used to evaluate energy harvesting systems is crucial, because testing with a wide range of program behaviors and truly long-running programs is more likely to expose an approach's tradeoffs.

Unless otherwise noted, we compile all benchmarks with `-O2` as the optimization level. We choose the `-O2` level since it includes most optimizations while avoiding code size versus speed tradeoffs. As Section 5.3 illustrates, Ratchet supports all of LLVM's optimization levels. We use an average lifetime of 100 ms to match the setup of previous works. With an average lifetime of 100 ms running with a 24 MHz clock, this gives us a mean lifetime of 2,400,000 cycles. Before each bout of execution, the simulator samples from a Gaussian whose mean is the desired mean lifetime (in cycles) and uses that value as the number of clock cycles to execute for before inducing the next power cycle. To simulate a power cycle, we clear the register values.

Given this experimental setup, we set out to answer several key questions about Ratchet:

[4]Some of these applications require input that is read in from a file. Since many energy-harvesting systems do not include an operating system or file system, we instead compile the input into the binary.

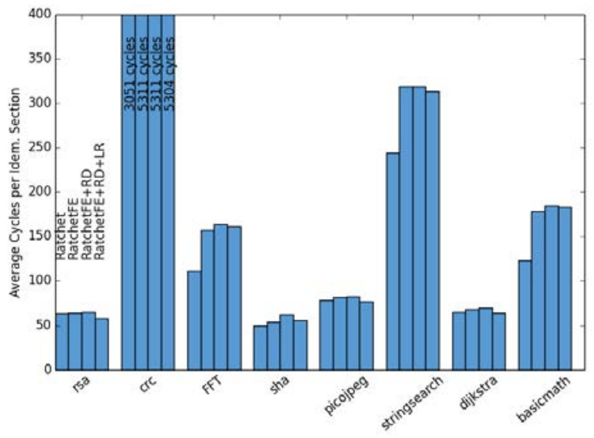


Figure 7: The average number of cycles per idempotent section break.

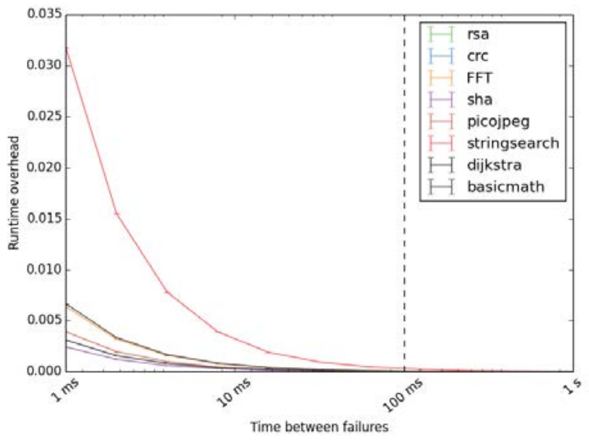


Figure 8: Re-execution overhead decreases as failure frequency increases. Note that CRC and RSA have zero re-execution overhead throughout since they are short enough to complete in a single power cycle.

1. Does Ratchet stretch computation across frequent, unpredictable losses of power correctly?
2. What is the overhead of running Ratchet due to checkpoints and re-execution?
3. Is Ratchet compatible with compiler optimizations?
4. What impact does Ratchet have on code size?

We use the results of this evaluation to compare Ratchet against alternative approaches that require hardware support or programmer intervention. See the results of this comparison in Table 1 and Section 7.2.

5.1 Performance

To understand the effects of power cycles on long-running programs and the overhead of Ratchet, we perform 10 trials of each benchmark with power failures as described earlier. Figure 6 displays the results of this experiment for each benchmark, averaged and normalized to the run time of the benchmark executed without power failures. The first thing to note is that 6 out of 8 of the benchmarks fail to complete execution on harvested energy without Ratchet (`w/o Ratchet`). There are several other results for each benchmark that represent successive Ratchet optimizations (all levels except `Ideal` maintain correctness): `Ratchet` shows the performance from our naive implementation; `RatchetFE` denotes placing a checkpoint at function entry only when there is a store to a non-local variable that is not preceded by a checkpoint; `RatchetFE+RD` is `RatchetFE`, but with duplicate checkpoints removed in LLVM's backend; `RatchetFE+RD+LR` adds a further optimization of only checkpointing the live-in registers; and finally, `Ideal` represents a lower bound on Ratchet's overhead that assumes perfect intraprocedural alias analysis and zero idempotence violations. `Ideal` bounds what is possible with more compiler engineering.

We observe an average run-time overhead of 58.9% using `Ratchet+FE+RD+LR`—a 20.1% improvement over `Ratchet`. The `Ideal` result suggests that further compiler engineering can reduce this overhead by over 60%. The total overhead includes run-time overhead due to saving checkpoints and re-execution, but checkpoint overhead dominates total overhead, because re-execution overhead approaches zero.

Looking at Figure 6, we can see that overhead varies dramatically between benchmarks. This shows that performance of our method is highly program dependent. Intuitively, this makes sense. If one program includes an implicit WAR dependence buried deep in the hot sections of code and another has very few WAR dependencies, we would expect their run times to vary dramatically. In order to determine the effect of idempotent section length of a benchmark on performance we measure the number of cycles between each checkpoint commit. To measure this, we instrument our simulator to measure the number of cycles[5] between each call to any of our checkpointing functions. We then run each benchmark to completion, without power cycles. The average number of cycles per idempotent section for each benchmark is shown in Figure 7. By comparing Figures 6 and 7 we notice that programs with shorter idempotent sections have higher overheads.

We also investigate the relationship between idempotent section length and re-execution overhead. To do this, we instrument our simulator to measure the number of

[5]Most instructions take a single clock cycle to complete in the ARMv6-M instruction set.

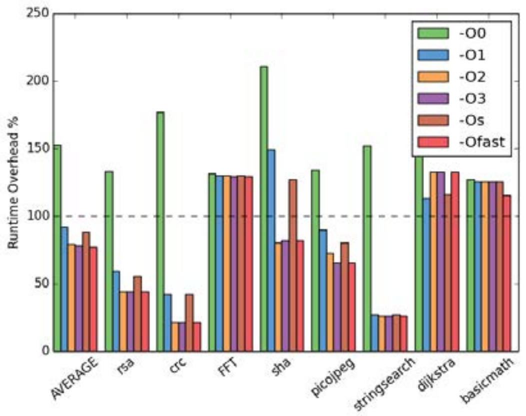


Figure 9: The runtime overhead of each benchmark compiled with Ratchet at various LLVM optimization levels normalized to the runtime of the uninstrumented benchmark compiled at `-O0`.

cycles from the last checkpoint to a checkpoint restore. This includes the cycles spent executing code that occurs after the last checkpoint and the cycles spent restarting and restoring the last checkpoint. Figure 8 shows the fraction of run-time overhead due to re-execution for a range of average power-on times. While short idempotent sections tend to cause higher overall overhead due to checkpoint overhead dominating the total overhead, Figure 8 combined with Figure 7 shows that benchmarks with shorter idempotent sections have lower re-execution costs. This is reasonable considering that a failure halfway through an idempotent section requires the program to re-execute from the last checkpoint. The more cycles since the last checkpoint, the higher the re-execution overhead. This suggests that increases in idempotent section length eventually will expose re-execution time as a key component of overhead.

5.2 Correctness

We validate Ratchet's correctness using both formal and experimental approaches. First, to test that Ratchet enforces idempotency with respect to non-volatile memory, we instrument the simulator to log reads and detect WAR dependencies that occur during execution. We consider a program to fail if there exists any load and subsequent store, to the same address, that are not separated by a checkpoint[6]. Second, to test that Ratchet enables long-running execution even with power-cycle-induced volatile state corruption, we simulate random power failures like those experienced in energy-harvesting devices

[6]This was especially helpful in debugging, as it exposed missed WAR dependencies, such as ones caused by spilling registers onto the stack due to register pressure, in early prototypes.

Program	Ratchet	Uninstrumented	Change
AVERAGE	563720	560824	1.79%
rsa	41326	40694	1.55%
crc	36037	34677	3.92%
FFT	182362	183612	-0.68%
sha	3286631	3284544	0.06%
picojpeg	379134	373051	1.63%
stringsearch	184656	177567	3.99%
dijkstra	183554	178465	2.85%
basicmath	216053	213978	0.96%

Table 2: Code size increase due to Ratchet (sizes are in bytes).

and verify the results. We check the validity by running different sequences of failures and hashing memory contents and registers at the completion of each benchmark run to compare the hash to the hash of the ground truth run without power failure. Lastly, to ensure Ratchet works even in the most energy starved environments we repeat this experiment with lifetimes as short as 1 ms.

5.3 Impact of Compiler Optimizations

In order to understand the performance of Ratchet under varying compiler optimizations, we benchmark Ratchet across each LLVM optimization level. Figure 9 shows the performance of the benchmarks compiled, with Ratchet, at different optimization levels relative to uninstrumented benchmarks compiled at `-O0`. We observe that in general, traditional compiler optimizations improve the performance of Ratchet. However, it also shows that in some cases, aggressive optimization results in higher perceived overheads. This suggests that breaking the program into idempotent sections can not only reduce the efficiency of optimizations, but also cause them to be detrimental (see Dijkstra).

5.4 Code size increase from Ratchet

Code size is a critical constraint for many energy-harvesting devices. In order to evaluate Ratchet's practicality with respect to code size, we measure the effect Ratchet has on code size. On average, Ratchet increases the size of the program by 1.79% or 2896 bytes. This increase is caused by adding our checkpoint recovery code, a number of optimized checkpointing functions, checkpoint calls throughout the program, and exchanging pop instructions for loads. Table 2 shows the change in code size for each benchmark. We notice that about 1356 bytes are a result of the additional function calls and reserved areas in memory. The rest of the code size increase can be attributed to inserted code or code that has been rewritten to support Ratchet, namely the translation of pop instructions. Note that the FFT program actually sees a decrease in size. We suspect this might be a result of Ratchet limiting optimization opportunities such as loop unrolling.

6 Discussion

There are several issues that represent corner-cases to Ratchet's design, such as avoiding repeating outputs on re-execution, instrumenting hand-written assembly, and dealing with the effects of power cycles too short to make forward progress. Instead of distracting from the core design, we discuss them here.

6.1 Output Commit Problem

In any replay-based system, there is a dilemma called the output commit problem, which states that a process should not send data to the outside world until it knows that that data is error free. The output commit problem takes on new meaning for energy harvesting devices; the problem is one of sending multiple outputs when only one should have been sent in a correct execution. This problem occurs during the re-execution of a code section that created an output during an earlier execution. Imagine an LCD interface on an embedded system, where there is re-execution while printing to the screen. We suggest placing checkpoints immediately before and after these output instructions to minimize the chance of re-execution due to the instruction being at the end of a long idempotent section. We believe that there is a wealth of future work on making protocols themselves robust against the effects of intermittent computation. One step in this direction is delay/disruption-tolerant networking [12].

6.2 Hand-Written Assembly

Hand-written assembly poses another challenge for Ratchet. Because it is never transformed into LLVM IR we cannot run our passes to determine if there are idempotency violations, and if so, where they occur. One naive approach is to simply checkpoint before and after the assembly. While that is a good start, there still remains the possibility of idempotency violations between loads and stores within a section of assembly. While we hand-process the assembly files required for newlib, and libgcc it is possible to create a tool to perform this processing automatically [8]. We choose not to write such a tool, since only three assembly files have idempotency violations. Instead we instrument those by hand.

6.3 Ensuring Completion

Extremely short power cycles pose a problem in that they prevent programs from making forward progress and thus never complete execution. Ratchet handles this problem both at compile time and at run time. At compile time, the programmer informs Ratchet of the expected on time for the device. Ratchet uses this information to make sure that no idempotent section is longer than the expected on time. Note that adding arbitrary checkpoints (*i.e.,* artificial idempotent section breaks) only affects overhead, *not* correctness. The second approach is to use the watchdog timer (commonly available on embedded systems) to insert checkpoints dynamically, when these quick power cycles prevent forward progress. That being said, the longest idempotent sections in our benchmarks (roughly 5000 cycles) require on-times of less than 0.2 ms to expose this issue.

7 Related Work

Ratchet builds upon previous work from three broad categories of research: rollback recovery, intermittently powered computing, and idempotence. We start by examining the history of general-purpose checkpointing and logging schemes, followed by how previous approaches to stretching program execution across frequent power cycles have so far adapted those general-purpose approaches. Lastly, since Ratchet builds upon previous work on idempotence, we cover that previous work.

7.1 Rollback Recovery

Research from the fault tolerance community presents the idea of backward error recovery, "backing up one or more of the processes of a system to a previous state which is hoped is error-free, before attempting to continue further operation" [36]. Backward error recovery systems often choose checkpointing as the underlying technique [11]. CATCH is an example of a checkpointing system that leverages the compiler to provide transparent rollback recovery [24].

The challenge of checkpointing is identifying and minimizing the state that needs to be protected by a checkpoint [36]. BugNet [31] eliminates the need for full-state checkpoints, while maintaining correctness, by logging the values loaded after an initial checkpoint. This represents a dramatic reduction in overhead as it is unlikely that a process reads the entire contents of memory. Revive [34] reduces overhead further (up to 50%) by using undo logging to record stores instead of loads—assuming a reliable memory.

Ratchet further reduces the bandwidth required by logging/checkpointing by extending idempotency to main memory. Idempotency tells us that only a subset of program stores actually require logging—those that earlier loads depend on. By checkpointing at stores that alias with earlier loads (since the last checkpoint), Ratchet is able to increase, by orders of magnitude, the number of instructions between log entries/checkpoints.

7.2 Intermittently Powered Computing

Research into fault tolerant computing traditionally targets persistently powered computers. However, new ultra-low-power devices challenge traditional power requirements, making them ideal candidates for energy harvesting techniques. Unfortunately, these techniques provide unreliable power availability that results in fragmented and incorrect execution—violating the assumptions of continuous power and infrequent errors. Because of this, previous work on intermittent computation adopts known rollback recovery techniques, but must adapt them to the properties of this new domain.

7.2.1 Hardware-assisted Checkpointing

Mementos [39] is the first system that attempts to tackle the intermittent computation problem through the use of a one-time (ideally) checkpoint. Mementos uses periodic voltage measurements of the system's energy storage capacitor to estimate how much energy remains. The goal is to checkpoint as late as possible. Mementos provides three ways to control the periodicity of voltage measurement: (1) function triggered: voltage measurement after every function return; (2) loop triggered: voltage measurement after every loop iteration; and (3) timer assisted: a countdown timer added to either of the first two variants that gates whether a voltage measurement is needed. Unfortunately, while Mementos works well when programs write to volatile state only, recent research shows that Mementos is incorrect in the general case [26, 38]. The problem is that Mementos allows for uncheckpointed work to occur. If uncheckpointed work updates both volatile and non-volatile memory, only the non-volatile memory will persist, creating an inconsistent software state. Applying the idea of Mementos to wholly non-volatile memory, as done by QuickRecall [17], actually makes the problem worse: in contrast to Flash-based systems, in systems with wholly non-volatile main memory, all program data is stored in non-volatile memory. This makes it a near certainty that a program will write to non-volatile memory after a checkpoint, leading to an inconsistent software state.

Hibernus [3] addresses QuickRecall's correctness issues through the introduction of guard bands. A guard band is a voltage threshold that represents the amount of energy required to store the *largest* possible checkpoint to non-volatile memory in the *worst-case* device and environmental conditions. Execution occurs while the voltage is above the threshold, but hibernates when the voltage is below the threshold. Doing this ensures that all work gets checkpointed, at the cost of wasting energy waiting for voltage to build up to the threshold and in the time after a non-worst-case checkpoint—there is no safe way to scavenge unused energy.

Hibernus also improves upon Mementos in terms of performance. Hibernus employs a more advanced analog-to-digital converter (ADC) (for voltage measurement) that allows software to set a threshold value that triggers an interrupt when the voltage goes below the threshold. This removes all software overhead caused by periodically checking the voltage (similar to polling versus interrupts in software).

Comparing voltage-triggered checkpointing systems to Ratchet is difficult. Fortunately, Hibernus provides a detailed performance comparison between itself and Mementos by implementing Mementos on their wholly non-volatile development platform. Their experimental results at 100ms on-time show that Mementos's total overhead varies between 117% and 145%, approximately, depending on the variant of Mementos. Hibernus's polling reduces total overhead to 38%, approximately. In comparison, Ratchet's overhead for the same benchmark program (potentially different inputs and configuration) is a comparable 32%.

In deciding between a one-time, voltage-triggered checkpointing scheme and Ratchet, the biggest factor is the ADC. Mandating an ADC is a non-starter for many existing systems that either do not have one already or systems that have one, but not connected in a way that supports power monitoring. This is a problem even for Mementos, "Voltage supervisors are common circuit components, but most—crucially, including existing prototype RFID-scale devices—-do not feature an adjustable threshold voltage." For future systems and those existing systems that have an ADC capable of voltage monitoring, even the most power efficient ADCs consume as much as 1/3 of the power of today's ultra-low-power microcontrollers [5]. The future is direr as performance/Watt tends to scale at 2x every 1.57 years for processors (known as Dennard scaling [10]), while ADC's performance/Watt tends to scale at 2x every 2.6 years [19].

7.2.2 Software-only Checkpointing

DINO [26], is a software-only, programmer-driven checkpointing scheme for tolerating an unreliable power source. Programmers leverage the DINO compiler to decompose their programs into a series of independent transactions backed by data versioning to achieve memory consistency in the face of intermittent execution. To accomplish this, the authors rely on programmer annotated logical tasks to define the transactions. Contrast this with Ratchet, which automatically decomposes programs using idempotency information inherent to a program's structure—allowing programmers to ignore the effects of power cycles and mixed volatility memory.

DINO enforces a higher-level property than does Ratchet, namely task atomicity, *i.e.,* tasks either complete or re-execute from the beginning, as viewed by other tasks on the system. This contrasts Ratchet, which allows intermediate writes as long as they maintain idempotency. Enforcing task atomicity happens to also solve the problem of intermittent computation (assuming the programmer defines short tasks), but enforcing a more restrictive property incurs more run-time overhead, 80% to 170%. Note that even though DINO is evaluated on a mixed-volatility system, these numbers are comparable to Ratchet's because DINO's overhead is more dependent on the amount of state a task may update than the volatility of that state.

7.3 Idempotence

Mahlke *et al.* develops the idea of idempotent code sections in creating a speculative processor [28]. The authors construct restartable code sections that are broken by irreversible instructions, which, "modifies an element of the processor state which causes intolerable side effects, or the instruction may not be executed more than one time." They use this notion to define how to handle a speculatively executing instruction that throws an exception and show that they can use the idempotence property to begin execution from the start of the interrupted section and still follow a valid control-flow path.

Kim *et al.* applies the idea of idempotency to data storage, showing that idempotency is useful for reducing the amount of data stored in speculative storage on speculatively multithreaded architectures [21]. The authors note that there are idempotent references that are independent of the memory dependencies that result in errors in non-parallizable code.

Encore is a software-only system that leverages idempotency for probabilistic rollback-recovery [13]. Targeted at systems using probabilistic fault detection schemes, Encore provides rollback recovery without dedicated hardware. The key insight of Encore is probabilistic idempotence: ignoring infrequently executed instructions that break idempotence can increase the length of idempotent sections. While this violates correctness—something Ratchet must maintain—the authors realized a performance improvement at the cost of not recovering 3% of the time.

De Kruijf *et al.* [7] presents an algorithm for identifying idempotent regions of code and show that it is possible to segment a program into entirely idempotent regions with minimal overhead. In this initial work, the authors focus their attention on soft faults that do not mangle the register state, noting that registers are usually protected by other means. While soft faults may not corrupt register state, power failures cause the entire register file to be lost.

In follow-on work, de Kruijf *et al.* [6] presents algorithms that utilize the idempotence information generated by the idempotent compiler to better inform the register allocation step of compilation. This allows the compiler to extend the live range of registers. Extending the live range of register values that are live-in to a given idempotent section all the way to the end of that section creates a free checkpoint that enables recovery from side-effect free faults. In contrast, faults on energy harvesting come with significant side effects.

8 Conclusion

Ratchet is a compiler that *automatically* enables the correct execution of long-running applications on devices that run on harvested energy, *without* hardware support. Ratchet leverages the notion of idempotence to decompose programs into a series of checkpoint-connected, re-executable sections. Experiments show that Ratchet stretches program execution across random, frequent, power cycles for a wide range of programs—that would not be able to run completely otherwise—at a cost of less than 60% total run-time overhead. Ratchet's performance is similar to existing approaches that require hardware support or programmer reasoning. Experiments also show that, with more engineering, it is possible to reduce run-time overheads to around 20%.

Ratchet shows that it is possible for compilers to reason about frequent failures and volatile versus non-volatile memory in languages not designed with either in mind. Pushing these burdens on the compiler opens the door for non-expert programmers to code for energy harvesting devices.

Acknowledgment

We thank our shepherd Y. Charlie Hu for his guidance and the anonymous reviewers for their feedback and suggestions. This work was supported in part by C-FAR, one of the six SRC STARnet Centers, sponsored by MARCO and DARPA.

References

[1] compiler-rt runtime [computer software]. Retrieved from http://compiler-rt.llvm.org/, Mar 2015.

[2] ARM. *ARMv6-M Architecture Reference Manual*, Sept 2010.

[3] BALSAMO, D., WEDDELL, A., MERRETT, G., AL-HASHIMI, B., BRUNELLI, D., AND BENINI, L. Hibernus: Sustaining computation during intermittent supply for energy-harvesting systems. *IEEE Embedded Systems Letters 7*, 1 (2014), 15–18.

[4] BUETTNER, M., PRASAD, R., SAMPLE, A., YEAGER, D., GREENSTEIN, B., SMITH, J. R., AND WETHERALL, D. RFID sensor networks with the Intel WISP. In *Conference on Embedded Networked Sensor Systems* (2008), SenSys, pp. 393–394.

[5] DAVIES, J. H. *MSP430 Microcontroller Basics*, 1 ed. Elsevier Ltd., 2008.

[6] DE KRUIJF, M., AND SANKARALINGAM, K. Idempotent code generation: Implementation, analysis, and evaluation. In *International Symposium on Code Generation and Optimization* (2013), CGO, pp. 1–12.

[7] DE KRUIJF, M. A., SANKARALINGAM, K., AND JHA, S. Static analysis and compiler design for idempotent processing. In *Conference on Programming Language Design and Implementation* (2012), PLDI, pp. 475–486.

[8] DEBRAY, S., MUTH, R., AND WEIPPERT, M. Alias analysis of executable code. In *Symposium on Principles of Programming Languages* (1998), POPL, pp. 12–24.

[9] DEBRUIN, S., CAMPBELL, B., AND DUTTA, P. Monjolo: An energy-harvesting energy meter architecture. In *Conference on Embedded Networked Sensor Systems* (2013), SenSys, pp. 18:1–18:14.

[10] DENNARD, R. H., GAENSSLEN, F. H., RIDEOUT, V. L., BASSOUS, E., AND LEBLANC, A. R. Design of ion-implanted MOSFET's with very small physical dimensions. *IEEE Journal of Solid-State Circuits 9*, 5 (Oct 1974), 256–268.

[11] ELNOZAHY, E. N., AND ZWAENEPOEL, W. Manetho: Transparent roll back-recovery with low overhead, limited rollback, and fast output commit. *IEEE Transactions on Computers 41*, 5 (May 1992), 526–531.

[12] FARRELL, S., CAHILL, V., GERAGHTY, D., HUMPHREYS, I., AND MCDONALD, P. When TCP breaks: Delay and disruption tolerant networking. *IEEE Internet Computing 10*, 4 (July 2006), 72–78.

[13] FENG, S., GUPTA, S., ANSARI, A., MAHLKE, S. A., AND AUGUST, D. I. Encore: Low-cost, fine-grained transient fault recovery. In *International Symposium on Microarchitecture* (2011), MICRO, pp. 398–409.

[14] GUTHAUS, M., RINGENBERG, J., ERNST, D., AUSTIN, T., MUDGE, T., AND BROWN, R. MiBench: A free, commercially representative embedded benchmark suite. In *International Workshop on Workload Characterization* (2001), pp. 3–14.

[15] HICKS, M. Mibench port targeted at IoT devices. `https://github.com/impedimentToProgress/MiBench2`, 2016.

[16] HICKS, M. Thumbulator: Cycle accurate ARMv6-m instruction set simulator. `https://github.com/impedimentToProgress/thumbulator`, 2016.

[17] JAYAKUMAR, H., RAHA, A., AND RAGHUNATHAN, V. QUICKRECALL: A low overhead hw/sw approach for enabling computations across power cycles in transiently powered computers. In *International Conference on Embedded Systems and International Conference on VLSI Design* (2014), pp. 330–335.

[18] JOGALEKAR, A. Moore's law and battery technology: No dice. *Scientific American* (Apr 2013).

[19] JONSSON, B. E. A survey of A/D-converter performance evolution. In *International Conference on Electronics, Circuits, and Systems* (Dec 2010), ICECS, pp. 766–769.

[20] KAHN, J. M., KATZ, R. H., AND PISTER, K. S. J. Next century challenges: Mobile networking for smart dust. In *International Conference on Mobile Computing and Networking* (1999), MobiCom, pp. 271–278.

[21] KIM, S. W., OOI, C.-L., EIGENMANN, R., FALSAFI, B., AND VIJAYKUMAR, T. N. Exploiting reference idempotency to reduce speculative storage overflow. *ACM Trans. Program. Lang. Syst. 28*, 5 (2006), 942–965.

[22] LATTNER, C., AND ADVE, V. LLVM: A compilation framework for lifelong program analysis & transformation. In *International Symposium on Code Generation and Optimization* (2004), CGO, pp. 75–86.

[23] LEE, Y., KIM, G., BANG, S., KIM, Y., LEE, I., DUTTA, P., SYLVESTER, D., AND BLAAUW, D. A modular 1mm3 die-stacked sensing platform with optical communication and multi-modal energy harvesting. In *International Solid-State Circuits Conference Digest of Technical Papers* (2012), pp. 402–404.

[24] LI, C.-C., AND FUCHS, W. Catch-compiler-assisted techniques for checkpointing. In *International Symposium on Fault-Tolerant Computing* (1990), pp. 74–81.

[25] LOWELL, D. E., CHANDRA, S., AND CHEN, P. M. Exploring failure transparency and the limits of generic recovery. In *Symposium on Operating Systems Design & Implementation* (2000), OSDI.

[26] LUCIA, B., AND RANSFORD, B. A simpler, safer programming and execution model for intermittent systems. In *Conference on Programming Language Design and Implementation* (2015), PLDI, pp. 575–585.

[27] MA, K., ZHENG, Y., LI, S., SWAMINATHAN, K., LI, X., LIU, Y., SAMPSON, J., XIE, Y., AND NARAYANAN, V. Architecture exploration for ambient energy harvesting nonvolatile processors. In *International Symposium on High Performance Computer Architecture* (2015), HPCA, pp. 526–537.

[28] MAHLKE, S. A., CHEN, W. Y., BRINGMANN, R. A., HANK, R. E., MEI W. HWU, W., RAMAKRISHNA, B., MICHAEL, R., AND SCHLANSKER, S. Sentinel scheduling: a model for compiler-controlled speculative execution. *ACM Transactions on Computer Systems 11* (1993), 376–408.

[29] MIRHOSEINI, A., SONGHORI, E., AND KOUSHANFAR, F. Idetic: A high-level synthesis approach for enabling long computations on transiently-powered asics. In *International Conference on Pervasive Computing and Communications* (2013), PerComm, pp. 216–224.

[30] MONTESINOS, P., HICKS, M., KING, S. T., AND TORRELLAS, J. Capo: A software-hardware interface for practical deterministic multiprocessor replay. In *International Conference on Architectural Support for Programming Languages and Operating Systems* (2009), ASPLOS, pp. 73–84.

[31] NARAYANASAMY, S., POKAM, G., AND CALDER, B. BugNet: Continuously recording program execution for deterministic replay debugging. In *International Symposium on Computer Architecture* (2005), ISCA, pp. 284–295.

[32] Plank, J. S., Beck, M., Kingsley, G., and Li, K. Libckpt: Transparent checkpointing under Unix. In *USENIX Technical Conference* (1995), TCON, pp. 18–29.

[33] Pokam, G., Danne, K., Pereira, C., Kassa, R., Kranich, T., Hu, S., Gottschlich, J., Honarmand, N., Dautenhahn, N., King, S. T., and Torrellas, J. QuickRec: Prototyping an Intel architecture extension for record and replay of multithreaded programs. In *International Symposium on Computer Architecture* (2013), ISCA, pp. 643–654.

[34] Prvulovic, M., Zhang, Z., and Torrellas, J. ReVive: cost-effective architectural support for rollback recovery in shared-memory multiprocessors. In *International Symposium on Computer Architecture* (2002), ISCA, pp. 111–122.

[35] Qazi, M., Clinton, M., Bartling, S., and Chandrakasan, A. A low-voltage 1 Mb FRAM in 0.13 mu m CMOS featuring time-to-digital sensing for expanded operating margin. *IEEE Journal of Solid-State Circuits 47*, 1 (Jan 2012), 141–150.

[36] Randell, B., Lee, P., and Treleaven, P. C. Reliability issues in computing system design. *ACM Computer Surveys 10*, 2 (1978), 123–165.

[37] Ransford, B., Clark, S., Salajegheh, M., and Fu, K. Getting things done on computational RFIDs with energy-aware checkpointing and voltage-aware scheduling. In *Conference on Power Aware Computing and Systems* (2008), HotPower, pp. 5–10.

[38] Ransford, B., and Lucia, B. Nonvolatile memory is a broken time machine. In *Workshop on Memory Systems Performance and Correctness* (2014), MSPC, pp. 5:1–5:3.

[39] Ransford, B., Sorber, J., and Fu, K. Mementos: System support for long-running computation on RFID-scale devices. In *International Conference on Architectural Support for Programming Languages and Operating Systems* (2011), ASPLOS, pp. 159–170.

[40] Raoux, S., Burr, G. W., Breitwisch, M. J., Rettner, C. T., Chen, Y.-C., Shelby, R. M., Salinga, M., Krebs, D., Chen, S.-H., Lung, H.-L., and Lam, C. H. Phase-change random access memory: A scalable technology. *IBM J. Res. Dev. 52*, 4 (July 2008), 465–479.

[41] Silicon Labs. EFM32ZG-STK3200 zero gecko starter kit.

[42] Texas Instruments. *MSP430FR59xx Datasheet*, 2014.

[43] Texas Instruments. MSP432P401x Mixed-Signal Microcontrollers, 2015.

[44] Van Der Woude, J., and Hicks, M. Ratchet source code repository. `https://github.com/impedimentToProgress/Ratchet`, 2016.

[45] Vinschen, C., and Johnston, J. Newlib [computer software]. Retrieved from https://sourceware.org/newlib/, Mar 2015.

[46] Wang, J., Dong, X., and Xie, Y. Enabling high-performance lpddrx-compatible mram. In *Proceedings of the 2014 International Symposium on Low Power Electronics and Design* (New York, NY, USA, 2014), ISLPED '14, ACM, pp. 339–344.

[47] Wu, Y.-C., Chen, P.-F., Hu, Z.-H., Chang, C.-H., Lee, G.-C., and Yu, W.-C. A mobile health monitoring system using RFID ring-type pulse sensor. In *International Conference on Dependable, Autonomic and Secure Computing* (2009), pp. 317–322.

[48] Xu, M., Bodik, R., and Hill, M. A "flight data recorder" for enabling full-system multiprocessor deterministic replay. In *International Symposium on Computer Architecture* (2003), ISCA, pp. 122–135.

[49] Zhai, B., Pant, S., Nazhandali, L., Hanson, S., Olson, J., Reeves, A., Minuth, M., Helfand, R., Austin, T., Sylvester, D., and Blaauw, D. Energy-efficient subthreshold processor design. *Transactions on Very Large Scale Integration Systems 17*, 8 (Aug 2009), 1127–1137.

[50] Zhang, H., Gummeson, J., Ransford, B., and Fu, K. Moo: A batteryless computational RFID and sensing platform. Tech. Rep. UM-CS-2011-020, Department of Computer Science, University of Massachusetts Amherst, Amherst, MA, 2011.

[51] Zhang, W., de Kruijf, M., Li, A., Lu, S., and Sankaralingam, K. ConAir: Featherweight concurrency bug recovery via single-threaded idempotent execution. In *International Conference on Architectural Support for Programming Languages and Operating Systems* (2013), ASPLOS, pp. 113–126.

Machine-aware Atomic Broadcast Trees for Multicores

Stefan Kaestle, Reto Achermann, Roni Haecki, Moritz Hoffmann, Sabela Ramos, Timothy Roscoe
Systems Group, Department of Computer Science, ETH Zurich

Abstract

The performance of parallel programs on multicore machines often critically depends on group communication operations like barriers and reductions being highly tuned to hardware, a task requiring considerable developer skill.

Smelt is a library that automatically builds efficient inter-core broadcast trees tuned to individual machines, using a machine model derived from hardware registers plus micro-benchmarks capturing the low-level machine characteristics missing from vendor specifications.

Experiments on a wide variety of multicore machines show that near-optimal tree topologies and communication patterns are highly machine-dependent, but can nevertheless be derived by Smelt and often further improve performance over well-known static topologies.

Furthermore, we show that the broadcast trees built by Smelt can be the basis for complex group operations like global barriers or state machine replication, and that the hardware-tuning provided by the underlying tree is sufficient to deliver as good or better performance than state-of-the-art approaches: the higher-level operations require no further hardware optimization.

1 Introduction

This paper addresses the problem of efficiently communicating between cores on modern multicore machines, by showing how near-optimal tree topologies for point-to-point messaging can be derived from online measurements and other hardware information.

The problem is important because parallel programming with message-passing is increasingly used inside single cache-coherent shared-memory machines, modern safe concurrent programming languages, runtime systems, and for portability between single-machine and distributed deployments.

This in turn leads to the need for distributed coordination operations, e.g. global synchronization barriers or agreement protocols for ensuring consistency of distributed state, to be implemented over message-passing channels. Efficient use of these channels becomes critical to program performance.

The problem is hard because modern machines have

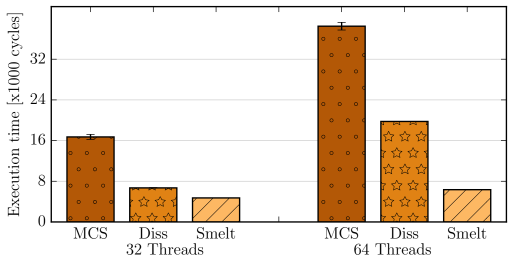


Figure 1: Comparison of thread synchronization using different barriers on Intel Sandy Bridge 4x8x2 with and without Hyperthreads. Standard error is < 3%.

complex memory hierarchies and interconnect topologies. Significant latency improvements for group operations can result from careful layout and scheduling of messages. Unlike classical distributed systems, the extremely low message propagation times within a machine mean that small changes to message patterns and ordering have large effects on coordination latency. Worse, as we show in our evaluation (§5.1), different machines show radically different optimal layouts, and no single tree topology is good for all of them.

In response, we automatically *derive* efficient communication patterns tuned for each particular machine based on online measurements of the hardware and data from hardware discovery. We realize this technique in Smelt, a software library which builds efficient multicast trees without manual tuning. Smelt serves as a fundamental building block for higher-level operations such as atomic broadcast, barriers and consensus protocols.

Smelt provides significant performance gains. Despite other modern barrier operations being highly tuned monolithic implementations, Smelt barriers are constructed over the multicast tree without the need for further hardware-specific tuning. Even so, they provide 3× better performance than a state-of-the-art shared-memory dissemination barrier, and is up to 6× faster than an MCS-based barrier (Figure 1) in our experiments (§5).

In the next section we further motivate this problem, and discuss the unexpected challenges that arise in solving it efficiently on real hardware platforms. We then describe Smelt's design (§3) and give details on its implementation (§4). We evaluate Smelt with a set of micro-benchmarks, existing runtime systems and applications in §5.

2 Motivation and background

We motivate our work in this section by first surveying the trend of building parallel programs using message-passing rather than shared-memory synchronization, even in a single cache-coherent machine. We then discuss messaging in a multicore machine, and how efficient implementations of group communication operations depend critically on the characteristics of complex and diverse memory systems. We then survey common tree topologies used in large computers for communication as a prelude to our discussion of Smelt in the next section.

2.1 The move to message passing

Modern high-end servers are large NUMA multiprocessors with complex memory systems, employing cache-coherency protocols to provide a consistent view of memory to all cores. Accessing shared data structures (e.g. shared-memory barriers or locks) thus entails a sequence of interconnect messages to ensure all caches see the updates [9]. This makes write-sharing expensive: cache lines must be moved over the interconnect incurring latencies of 100s of cycles. Atomic instructions like compare-and-swap introduce further overhead since they require global hardware-based mutual exclusion on some resource (such as a memory controller).

This has led to software carefully laying out data in memory and minimizing sharing using techniques like replication of state, in areas as diverse as high-performance computing [32], databases [35], and operating systems [3]. Systems like multikernels eschew shared-memory almost entirely, updating state through communication based on message-passing.

There are other reasons to use local message-passing. Several modern systems languages either have it as a first-class language feature (like Go) or impose strong restrictions on sharing memory access (like Rust).

Furthermore, while contemporary machines are mostly coherent, future hardware might not even provide globally shared *non*-coherent memory [13]. Here, efficient message-passing is not merely a performance optimization – it is required functionality. The same is true today for programs that span clusters of machines. A single paradigm facilitates a range of deployments.

Ironically, most NUMA message-passing mechanisms today use cache-coherence. With few exceptions [4], multicore machines provide no message-passing hardware. Explicit point-to-point message channels are implemented above shared-memory such that a cache line can be transferred between caches with a minimum number of interconnect transactions. Examples are URPC [5], UMP [2] and FastForward [16]; in these cases, only two threads (sender and receiver) access shared cache lines.

2.2 Communication in multicores

While cache-coherency protocols aim to increase multicore programmability by hiding complex interactions when multiple threads access shared-memory, this complexity of the memory hierarchy and coherency protocol makes it hard to reason about the performance of communication patterns, or how to design near-optimal ones.

The protocols and caches also vary widely between machines. Many enhancements to the basic MESI protocol exist to improve performance with high core counts [18, 28], and interconnects like QPI or HyperTransport have different optimizations (e.g., directory caching) to reduce remote cache access latency.

Worse, thread interaction causes performance variabilities that prevent accurate estimation of communication latency. For example, when one thread polls and another writes the same line, the order in which they access the line impacts observed latency.

Prior work characterized [29] and modelled [32] coherence-based communication, optimizing for group operations. However, these models require fine-grained benchmarking of the specific architectures, providing more accurate models but less portable algorithms.

Smelt is much more general: we abstract coherence details and base our machine model on benchmark measurements, simplifying tree construction while still adapting to underlying hardware. We show that sufficient hardware details can be obtained from a few microbenchmarks which are easily executed on new machines without needing to understand intricate low-level hardware details.

2.3 Group communication primitives

In practice, message-passing is a building block for higher-level distributed operations like atomic broadcast, reductions, barriers or agreement. These require messages to be sent to many cores, and so they must be sent on multiple point-to-point connections.

The problem described above is therefore critical, particularly in complex memory hierarchies. A large coherent machine like an HP Integrity Superdome 2 Server has hundreds of hardware contexts on up to 32 sockets, with three levels of caching, many local memory controllers, and a complex interconnection topology.

Consider a simple broadcast operation to all cores. Baumann et *et al.* [2] show how a careful tree-based approach to broadcast outperforms and out-scales both sequential sends *and* using memory shared between all recipients. Both the topology and the *order* to send messages to a node's children are critical for performance.

The intuition is as follows: unlike classical distributed systems, message propagation time in a single machine is negligible compared to the (software) send- and receive time as perceived by the sender and receiver. The store operation underlying each send operation typically takes a few hundred cycles on most machines. Loads on the receiver wait until the cache line transfer is done, which is between 300 and 1000 cycles depending on the machine. Since sequential software execution time therefore dominates, it is beneficial to involve other cores quickly in the broadcast and exploit the inherent parallelism available. Figure 2 shows an example of such a hierarchical broadcast in a 8-socket machine.

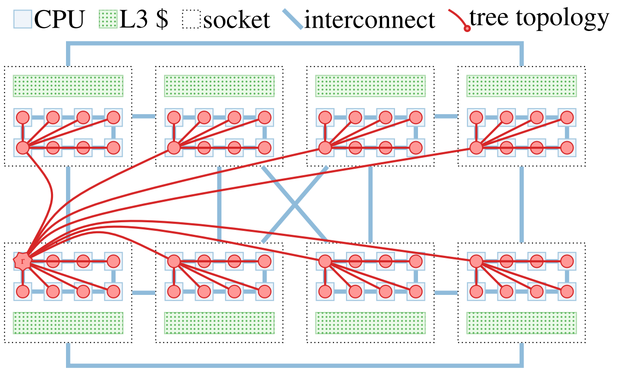


Figure 2: Multicore with message-passing tree topology

The cost of sending and receiving messages between cores is a subtle machine characteristic typically not known to programmers, and depends both on the separation of cores in the machine hierarchy and on more complex subtleties of the cache-coherency protocol. Very little of this information is provided by hardware itself (e.g. in the form of registers with hardware features) and vendor specifications are often incomplete or vague. Worse hardware *diversity* is increasing as much as complexity.

Despite this, prior work has built machine-optimized broadcast trees (e.g. Fibonacci trees [8]), and MPI libraries provide shared-memory optimizations for group or collective operations which try to account for NUMA hierarchies [25] using shared-memory communication channels [17]. Our results in §5 show that these trees are sometimes good, but there is no clear winner across all our machines we used for evaluation.

2.4 Common tree topologies

We now introduce the tree topologies we evaluate in this paper. The first three are hardware-oblivious, constructed without accounting for the underlying topology. We provide a visualization for each of these on all evaluation machines: http://machinedb.systems.ethz.ch/topologies ↗[1].

Binary trees have each node connected to at most two children. Here, each node n connects to nodes $2n + 1$ and $2n + 2$. Such trees often introduce redundant cross-NUMA links, causing unnecessary interconnect traffic. Performance is suboptimal since the low fanout (two) often means that nodes become idle even when they could further participate in the protocol. ↗

Fibonacci trees [22] are widely used unbalanced trees: left-hand subtrees are larger than right-hand ones. In contrast to binary trees, this imbalance allows more work to be executed in sub-trees that receive messages earlier, which prevents nodes from being idle when they could otherwise further participate in the broadcast. However, like binary trees they also have a fixed fanout and can exhibit redundant cross-NUMA transfers. ↗

Sequential trees are also widely used: a root sends a message to each other node sequentially (star-topology). Send operations are not parallelized since one node does all the work. This scales poorly for broadcasts on most large multicore machines. ↗

The other three trees we compare with consider machine characteristics in their construction:

Minimum spanning trees (MSTs) use Prim's algorithm [31] by adding edges in ascending order of cost until the graph is connected. This minimizes expensive cross-NUMA transfers, but does not optimize fanout and hence send parallelism. Among others, the resulting topology can be a star or a linear path, which both are purely sequential in sending. ↗

Cluster trees are built hierarchically, as in HMPI [25]. A binary tree is built between NUMA nodes, and messages are sent sequentially within a node. ↗

Bad trees are a worst-case tree example, built by running an MST algorithm on the inverse edge costs, maximizing redundant cross-NUMA links. We use this to show that the topology matters, and choosing a sub-optimal tree can be as bad as sequentially sending messages on some machines. ↗

3 Design

We now elaborate on the design considerations for Smelt and describe our multicore machine model (§3.1) including its properties and assumptions. Next, we show how we populate it (§3.2) and how our adaptive tree is built (§3.3). In §3.4, we will further show that exhaustive search is not a viable solution for finding the optimal broadcast tree.

Smelt is a library which simplifies efficient programming of multicore machines by providing optimized atomic broadcast trees adapted to the hardware at hand. The tree topology and sending order is generated automatically from a machine model constructed from information given by the hardware itself and a set of fine-grained micro-benchmarks.

Surprisingly, the trees derived by Smelt also function

[1]We refer to http://machinedb.systems.ethz.ch with the symbol ↗

well as building blocks for higher-level protocols – for example, given a Smelt tree, an efficient barrier can be implemented in two lines of code and performs as well as or better than state-of-the-art shared-memory barriers written and hand-tuned as monolithic components.

As a further evaluation of the applicability of Smelt, we also re-implemented the 1Paxos [10] consensus algorithm using Smelt trees (§5.6), which can be used to provide consistent updates on replicated data. Further, we build a key-value store on top of that (§5.7), which provides good performance for consistently replicated data.

Current server machines, of course, do not exhibit partial failure, and so our agreement protocol should be considered a proof-of-concept rather than a practical tool. However, we note that even in the absence of failures, replication within a multicore can be useful for performance [21, 35]. Coordination and synchronization would then still be needed to provide a consistent view of the system state. However, it seems likely that future machines will expose partial failures to programmers [13], requiring replication of services and data also for fault-tolerance.

Smelt combines various tools to a smart runtime system: (*i*) a machine model with micro-benchmarks to capture hardware characteristics, (*ii*) a tree generator for building optimized broadcast trees and (*iii*) a runtime library providing necessary abstractions and higher-level building blocks to the programmer. We visualize this in Figure 3.

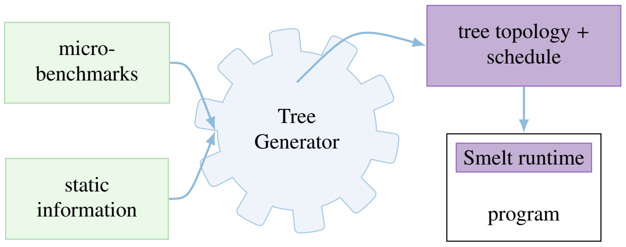


Figure 3: Overview of Smelt's design

3.1 Modelling broadcasts on multicore

Figure 4 visualizes a timeline for message-passing on a multicore system, specifically the time that it takes for a thread v_i to send a message to two threads v_j and v_k.

Unlike classical networks, t_{send} and $t_{receive}$ times dominate the total transmission time. We show this in our pairwise send and receive time (§3.2). This is significant as the sending and receiving threads are blocked for t_{send} and $t_{receive}$ respectively. It implies that the cost of sending n messages grows linearly with the number of messages to be sent, whereas in classical distributed systems, $t_{propage}$ dominates independently of how many messages are sent in a single round trip.

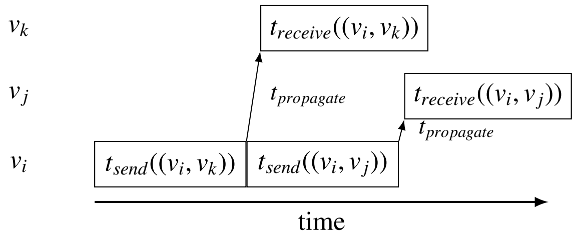


Figure 4: Visualization of a send operation: thread v_i sends a message to v_k followed by another message to v_j. Send operations are sequential, while the receive operations can be processed in parallel on threads v_j and v_k.

Multicore machine model: Communication in a multicore machine can be represented as a fully connected graph $G = (V, E)$. Vertices v_i correspond to threads, and edges $e = (v_i, v_j)$ model communication between threads v_i and v_j, with edge weights as a tuple $w(e) = (t_{send}, t_{receive})$. We show an example of such a graph in Figure 5. We now define t_{send}, $t_{receive}$ and $t_{propage}$:

$t_{send}(e)$ denotes the time to send a message over an edge $e = (v_i, v_j)$ from sender v_i to receiver v_j. The sender v_i is blocked during this time. Sending a message involves invalidating the cache line to be written and therefore often depends on which cores the cache line is shared with and also depend on the sequence of previously sent messages at the sender. Moreover, it may vary with the state of the cache line to be written.

$t_{receive}(e)$ denotes the time to receive an already-queued message. The receiver is blocked while receiving the message. In many cache-coherency protocols, receiving (reading) changes the state of the cache line in the receiver's cache from invalid to shared, and from modified to shared in the sender.

$t_{propagate}(e)$ is the time it takes to propagate a message on the interconnect. Propagation time is neither visible on the sender nor receiver. We assume $t_{propagate} = 0$, as propagation time can be seen as part of the $t_{receive}$.

Our model is similar to the telephone model [38], with a few differences: In the telephone model, each participant has to dial other participants sequentially before transmitting data. Similarly, the Smelt model has a sequential component when sending: the thread is blocked for the duration of t_{send}, and consecutive sends cannot be executed until the previous ones are completed. However, note that several threads can send messages concurrently and independently of each other.

The weight of edges in our model is non-uniform: the cost of receiving and sending messages from and to cores that are further away (NUMA distance) is higher. This

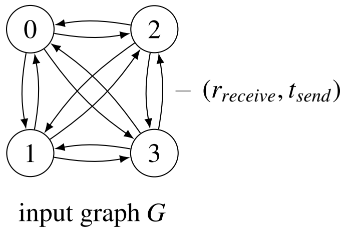


Figure 5: Example of fully connected input graph for four CPUs and one possible resulting broadcast tree topology τ with root 1 and send order as edge weights.

is even true for sending messages, since cache lines may have to be invalidated on the receiver before they can be modified on the sender.

Output The desired output consists of three parts: (*i*) A root v_{root}, (*ii*) a tree $T = (V, E')$ with $E' \subseteq E$, where T is a spanning tree of G, (*iii*) edge priorities $w'(v)$ representing the order in which a vertex v sends messages to its children. T is to be chosen such that the latency $lat(T)$ is minimal. The latency is given recursively as

$$lat(T) = \max(\forall v \in V : lat(T, v))$$

with

$$lat(T, v_{root}) = 0$$
$$lat(T, v) = lat(T, v_p) + \sum_{i=1}^{k} t_{send}(v_p, v_i)$$

with v_p being parent of v and v the k-th child of v_p. Note that this latency includes the send-cost for $k-1$ children that v_p sends a message to first.

3.2 Populating the machine model

We derive the input values for our machine model from various sources: libraries such as `libnuma` [36], tools like `likwid` [34], or special OS provided file systems like `/proc` and `/sys` on Linux. The OS and libraries obtain their information by parsing ACPI [20] tables like the static resource affinity table (SRAT) and system locality information table (SLIT), which, for example, provide the NUMA configuration. However, this information is coarse-grained and insufficient for our purposes.

To address this, Smelt enriches relevant static machine information with a carefully chosen set of micro benchmarks that capture relevant hardware details that cannot be inferred from this static information.

Pairwise send and receive time (t_{send} and $t_{receive}$) Motivated by §3.1, we measure the pairwise send (t_{send}) and receive ($t_{receive}$) latency between all hardware threads in the system. Figure 6 shows a visualized output of this benchmark on a 32-core AMD machine (A IL 4x4x2 in Table 1). Both the receive and send time clearly show the NUMA hierarchy of this eight node system. Note that the receive costs are asymmetric: $t_{receive}$ does not only depend on the NUMA distance but also on the direction i.e.

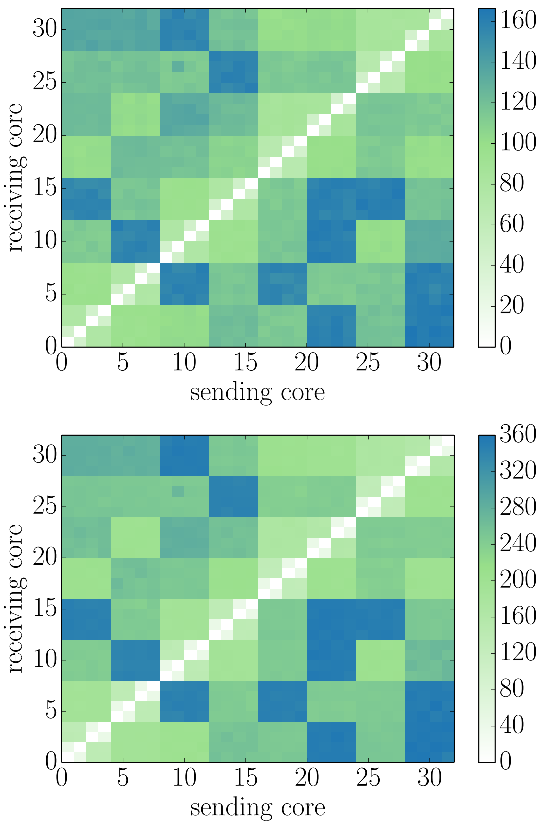


Figure 6: Pairwise send (upper) and receive time (lower) on A IL 4x4x2.

$t_{receive}(v_i, v_k) \neq t_{receive}(v_k, v_i)$. This observation is inline with [24].

The send cost is smaller compared to receive and (on this machine) shows the same NUMA hierarchy. Note that we measure the cost of sending batch of 8 messages and take the average to compensate for the possible existence of a hardware write buffer: software is only blocked until the store is buffered (and not until the cache line is fetched). This hides the full cost of the cache-coherency protocol. Ideally, Smelt would measure the effect of the write buffer as well. This makes benchmarking significantly more complex as the cost of sending a message between two cores would also depend on the cost of previous send operations and the ocupancy of the write buffer.

Fortunately, write buffers are relatively small, so that their effects do not change the runtime behavior significantly. We keep such a benchmark open for future work and use the more simple batch sending approach instead.

Data from our pairwise send- and receive experiments allow us to predict the time a core is busy sending or receiving a message and when it will become idle again. The busy/idle pattern of the cores is essential to decide which topology to use and the send-order of messages. Our pairwise benchmark works independently of the cache-coherency protocol as it determines the cost of sending and receiving messages using the same software-message-passing library and hence including all overheads induced by interconnect transfers triggered by the cache-coherency protocol implemented on each machine.

Pairwise benchmarks for all our machines are available on our website (http://machinedb.systems.ethz.ch) and the pairwise benchmark's code can be found on http://github/libsmelt.

3.3 Tree generation: adaptive tree

We now describe how Smelt generates trees automatically based on the machine model. Since the tree topology for multicast operations depends on which cores are allocated for an application, a new tree has to be generated whenever an application is started. Smelt generates the tree topologies offline in a separate tree generation step. Currently, this is done in a separate process when the application is initialized, but we anticipate that generating a tree topology will be a frequent operation on future machines as a consequence of reconfiguration in the case of failure or if group communication membership changes. We show the tree topologies generated for each of our machines on http://machinedb.systems.ethz.ch ↗.

3.3.1 Base algorithm

When building our adaptive tree, we rely on the performance characteristics described in §3.2, and make use of the fact that the tree generator can defer a global view of the system state from message send and receive times. For example, it knows which messages are in transit and which nodes are idle. The tree generator operates as an event-based simulation using our pairwise measurements. Whenever a core is idle, it uses the model to choose the next core to send the message to.

The desired output of the tree generator is (*i*) the root of the tree, (*ii*) a spanning tree connecting all nodes, and (*iii*) a schedule that describes the send-order in each node. If not specified by the application, we select the root to be the thread having the lowest average send cost to every other node in the system. The task then is to design a good heuristic to find near-optimal solutions for finding the tree topology. For example, messages can first be sent on expensive links to minimize the cost of the link that dominates the broadcast execution time. Alternatively, we can send on cheap links first to increase the level of parallelism early up. We found that for current multicore machines, it is more important to send on expensive links first: local communication is comparably fast, so messages can be distributed locally and efficiently once received on a NUMA node.

Another trade-off is between minimizing expensive cross-NUMA links and avoiding nodes being idle. For the machines we evaluated, we found that there is little or no benefit from sending redundant cross-NUMA messages even if cores are otherwise idle. For the few exceptions, our optimizations described in §3.3.2 detect opportunities for additional cross-NUMA links and adds them iteratively later where appropriate. Our heuristics are as follows:

- Remote cores first: We prioritize long paths as they dominate the latency for broadcasts. We select the cheapest core from the most expensive remote NUMA node. Local communication is executed afterwards, since this is relatively cheap.
- Avoid expensive communication links: We send the message to a remote NUMA node only if no other core on that node has received the message. We can do this because our tree generator has global knowledge on the messages in flight. This minimizes cross-NUMA node communication.
- No redundancy: We never send messages to the same core twice. The tree generator knows which messages are in flight and will not schedule another send operation to the same core.
- Parallelism: We try to involve other nodes in the broadcast as much as possible. The challenge here is to find the optimal fan-out of the tree in each node. The result often resembles an imbalanced tree so that cores that received a copy of the message early have a larger sub-tree than later ones.

We describe the tree generation in detail in Algorithm 1. At any point during the generation run, a core in the tree generator can be in either of two states: (*i*) *active* meaning that it has received a message and is able to forward message to other nodes, or (*ii*) *inactive* otherwise. Inactive nodes are waiting to receive a message from their parent. The set of active cores is denoted as A_{cores}. NUMA nodes are active if at least one of its cores is active (A_{nodes}).

We observe that the tree obtained from this algorithm is a multilevel tree for most machines (↗). Message delivery is first executed across NUMA nodes and then further distributed within each node. The tree generator creates a multilevel hierarchy in either of these steps only if the send operation is relatively expensive compared to receive operations. Otherwise, it will sequentially send messages. For example, with a NUMA node, due to relatively low send costs, sequentially sending messages is often faster than a multilevel sub-tree, especially for systems with only a few cores per node.

In our evaluation (§5.1), we show that a tree generated with our tree generator performs comparably with or better than the best static tree topology on a wide range of machines. While the algorithm itself might have to be adapted in the future to cope with changes in hardware development, the approach of using micro bench-

Algorithm 1 Smelt's adaptive tree

```
$C_{all}$                                          ▷ Set of cores
$A_{nodes} \leftarrow$ NODE_OF($c_{root}$)                    ▷ Active nodes
$A_{cores} \leftarrow c_{root}$                                ▷ Active cores
function PICK_MOST_EXPENSIVE($C$, $i$)
    return $\arg\max_{x \in C} (t_{send}(i,x) + t_{receive}(i,x))$
end function
function PICK_CHEAPEST($C$, $i$)
    return $\arg\min_{x \in C} (t_{send}(i,x))$
end function
function NODE_IDLE($i$)   ▷ Executed for active idle node $i$
    $C_{inactive} \leftarrow C_{all} \cap (A_{cores} \cup$ CORES_OF($A_{nodes}$) )
    $c_{next} \leftarrow$ PICK_MOST_EXPENSIVE($C_{inactive}$, $i$)
    if SAME_NODE($i$, $c_{next}$) then                     ▷ Local
        SEND($c_{next}$)
        $A_{cores} \leftarrow A_{cores} \cup c_{next}$              ▷ Mark core active
    else
        $C_{eligible} \leftarrow$ NODE_OF($c_{next}$)          ▷ All cores on node
        $c_{next} \leftarrow$ PICK_CHEAPEST($C_{eligible}$, $i$)
        SEND($c_{next}$)                               ▷ Send remotely
        $A_{nodes} \leftarrow A_{nodes} \cup$ NODE_OF($c_{next}$)
        $A_{cores} \leftarrow A_{cores} \cup r$
    end if
end function
function ADAPTIVE_TREE                         ▷ Run for core $c_{self}$
    while $C_{all} \cap A_{cores} \neq \emptyset$ do
        if $c_{self} \in A_{cores}$ then
            NODE_IDLE($c_{self}$)
        else
            WAIT_MESSAGE
        end if
    end while
end function
```

marks to capture fine-grained hardware details for building machine-aware broadcast trees should still be applicable. Programmers then automatically benefit from an improved version of the algorithm constructing the tree, even in the presence of completely new and fundamentally different hardware without having to change application program code. Consequently, we believe our generator to be useful for future increasingly heterogeneous multicores.

Note that our algorithm is designed for broadcasts trees, but we show in §5 that it also works well for reductions. However, our design and implementation are flexible enough to use different trees for reductions and broadcasts if necessary for future hardware.

3.3.2 Incremental optimization

Smelt's base algorithm produces an hierarchical tree, where only one expensive cross-NUMA link is taken per node. This gives a good initial tree, but leaves room for further improvements, which we describe here:

Reorder sends: most expensive subtree Smelt's basic algorithm as described before sends on expensive links first. This is a good initial strategy, but can be further improved after constructing the entire tree-topology. In order to minimize the latency of the broadcast, the time until a message reaches the last core has to be reduced. Sending on links that have the most expensive sub-tree intuitively achieves that.

Shuffling: adding further cross-NUMA links As soon as a NUMA node is active, i.e. has received a message or has a message being sent to it already, it will not be consider for further cross-NUMA transfers. On larger machines, this can lead to an imbalance, where some threads already terminate the broadcast and become idle when they could still further participate in forwarding the message to minimize global latency of the broadcast tree.

Figure 7 shows this as a simple example for only two cores 0 and 14. Core 14 finishes early and does not consider sending any more messages, since each other NUMA node is already active and all its local nodes finished as well. Core 0 terminates considerably later. The time between core 0 and core 14 finishing is t_{slack} as indicated in the figure.

If an additional cross-NUMA link between core 14's and core 10's NUMA node would terminate faster despite adding another expensive cross-NUMA link, it is beneficial from a purely-latency perspective to allow core 14 to execute this additional NUMA link replacing the link that initially connected node 0 before this optimization.

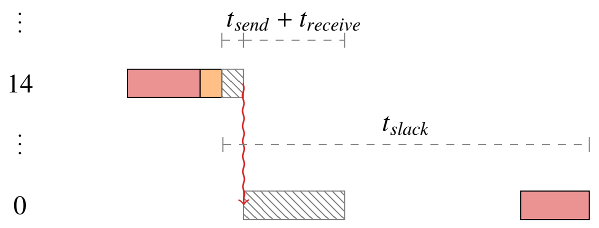


Figure 7: Optimization: add further cross-NUMA links

Smelt executes the following algorithm to decide based on the model if an additional cross-NUMA link $v_s \to v_e$ would reduce the latency of the broadcast. In each iterative step, we select nodes v_s and v_e as the node that first becomes idle and the node that terminates last respectively. If $t_{send} + t_{receive} < t_{slack}$, Smelt adds an additional cross-NUMA link. Then we iteratively optimize until adding edge $v_s \to v_e$ does not further reduce the latency of the tree. If this is the case, the resulting tree from replacing previous edge $v_x \to v_e$ with $v_s \to v_e$ is always better according to the model. If slower, the algorithm would not have chosen to optimize it and terminated.

The result can be further improved by sorting the edges. Hence, after each "shuffle"-operation, we reorder the scheduling of sends on each outgoing connection of a core by the cost of the receiving core's sub-tree as described in the previous section.

3.4 Finding the optimal solution

Despite being a type of minimum spanning tree, traditional graph algorithms cannot be used to solve the MST problem in our context as they do not consider the edge priorities. In fact, finding a broadcast tree in the telephone model for an arbitrary graph is known to be NP hard [37].

A brute-force approach to the problem is not feasible as the search space grows rapidly. To obtain the best tree given a set of nodes n, we need to construct first all the possible trees with n nodes. This is the Catalan Number [11] of order $n-1$ (C_{n-1}). Moreover, there are $n!$ possible schedules per tree. Hence, the number of possible configurations is shown in Equation 1.

$$Ntrees = n!C_{n-1} = \frac{(2n-2)!}{(n-1)!} \tag{1}$$

Assuming 1ms for evaluating the model of a single tree, this would take over 6 months for 10 cores. We ran a brute-force search for up to 8 cores (5 hours) for a subset of our machines, and compared the adaptive tree against the optimal solution under the model. The estimated runtime by our tree generator predicts that Smelt has a relative error of 9% versus the optimal solution. In real hardware the error is larger at around 13%.

Note that we designed our algorithm for large multicore machines, and evaluating its optimality for configurations of only 8 cores does not show the full potential of our methodology. Unfortunately, due to the high cost of calculating the optimal tree with a brute-force approach, we were not able to extend this validation to bigger machines.

4 Implementation

The Smelt runtime (SmeltRT) is a C++ library that allows the programmer to easily implement machine optimized higher-level protocols by abstracting the required channel setup and message-passing functionality. SmeltRT is structured in two layers: (*i*) a transport layer providing send/receive functionality and (*ii*) a collective layer supporting group communication. For each layer, we explain its core concepts, interfaces and abstractions.

4.1 Transport layer

SmeltRT uses message-passing as a communication mechanism between threads, which SmeltRT pins to cores. The transport layer provides point-to-point message-passing functionality between threads including `send()`, `receive()` and OS independent control procedures. Those message-passing channels are abstracted using a bi-directional *queuepair*, allowing different transport backends to be used transparently.

Properties All queuepair backends must implement the following properties: (*i*) reliability: a message sent over a queuepair is never lost or corrupted and will be received eventually. (*ii*) ordering: two messages sent over the same queuepair will be received in the same order. (*iii*) a queuepair can hold a pre-defined number of messages. Since today's multicores are reliable, only flow control has to be implemented to guarantee above properties. It is needed to notify the sender which slots can be reused for further messages.

Interface The basic send/receive interface can be seen in the following listing. Messages are abstracted using Smelt-messages which encapsulate payload and length to be sent over the queuepair. The send and receive operations may block if the queuepair is full or empty respectively. The state of a queuepair can be queried to avoid unecessary blocking.

```
errval_t
smlt_queuepair_send(struct smlt_qp *qp,
                    struct smlt_msg *msg);
errval_t
smlt_queuepair_recv(struct smlt_qp *qp,
                    struct smlt_msg *msg);
```

Message-passing backends The transport layer is modular and supports multiple backends. Each queuepair backend must adhere to the properties of a Smelt queuepair as stated above. Smelt's message-passing backend is an adapted version of the UMP channel used in Barrelfish [2], originally inspired by URPC [5]. A Smelt UMP queuepair consists of two circular, unidirectional buffers of cache line-sized message slots residing in shared-memory. Each message contains a header word which includes the sequence number for flow-control and an epoch bit to identify new messages. Each cache line has one producer and one consumer to minimize the impact of the cache-coherency protocol. The cache lines holding messages are modified only by the sender. The receiver periodically updates a separate cache line with the sequence number of the last received message. This line is checked by the sender to determine whether a slot can be reused.

Further, we implemented other shared-memory backends (such as FastForward [16]) and plan to support inter-machine message-passing backends over IP or RDMA protocols in the future.

4.2 Collective layer

The collective layer builds upon the transport layer and provides machine-aware, optimized group communication primitives such as broadcasts and reductions. A reduction is a gather operation which blocks until the node has received the value from all its children and the aggregate is forwarded to the parent. Such collective operations involve one or more threads in the system.

4.2.1 Concepts

Smelt's collective operations rely on Smelt's core concepts of *topologies* and *contexts*.

Topologies A topology describes the communication structure for the collective operation. It defines which participants are part of this communication group (i.e. multicast). Each topology has a distinct node, the root, where all broadcasts are initiated and all reductions end. The topology can be generated deterministically at runtime, loaded from a configuration file, or returned by a service. The latter two can describe a hardware aware topology either precalculated or supplied on demand respectively.

Contexts Smelt takes the topology description as a blueprint for creating the required transport links which are encapsulated in a context. A topology can be used to create multiple contexts. Collective operations require a valid context to identify the parent and child nodes for sending and receiving messages.

4.2.2 Collective operations

Next we show the interface for collective operations. The context identifies the location in the tree of the calling thread and defines the operations being executed. The reduction takes an `operation` argument – a function pointer – to implement the reduction operation. At the root, the `result` parameter contains the gathered value.

```
errval_t smlt_broadcast(struct smlt_context *ctx,
                        struct smlt_msg *msg);
errval_t smlt_reduce(struct smlt_context *ctx,
                     struct smlt_msg *input,
                     struct smlt_msg *result,
                     smlt_reduce_fn_t op);
```

Broadcasts Smelt's broadcast primitives guarantee reliability and ensure that all nodes in a given context receive messages in the same order (atomic broadcast). We assume that multicores today are fail-stop as a whole and hence either run reliable or the entire machine fails. Smelt's broadcasts start at a defined, per context root and therefore all messages are sent through the root, that acts as a sequentializer. It is possible to have multiple contexts with different roots. In that case, however, each core has to poll several memory locations for messages associated with multiple endpoints from different trees. This increases the receive overhead on each core, but also the latency, as multiple channels have to be polled. The sequentializer, together with the FIFO property of the edge links and reliable transmission, implements the atomic broadcast property.

Reductions Reductions do in-network processing on each node from payload received from all children, and pass the new value to the parent node. We use the same tree as in the broadcasts and the final value can be obtained at the root.

Barriers With the basic collective operations `reduce` and `broadcast`, we implement a `barrier` as shown in the code below. Note that we use the optimized zero-payload variants of `reduce` and `broadcast`. Despite its simplicity, our `barrier` outperforms or is comparable to state-of-the-art implementations, as we show in §5.4 and §5.5. This demonstrates that a highly-tuned generic machine-aware broadcast as implemented by Smelt can be used for higher-level protocols that benefit automatically from Smelt's optimizations.

```
void smlt_barrier(struct smlt_context *ctx) {
    smlt_reduce(ctx);
    smlt_broadcast(ctx);
}
```

5 Evaluation

We ran our experiments on eleven machines of two vendors with different microarchitectures and topologies (c.f. Table 1). Throughout this section, we indicate the number of threads as triple #sockets×#cores×#threads.

Due to space constraints, in most sections we focus on the largest machines for each vendor. In addition to the results shown here, we provide a website showing detailed results for all experiments on each machine: http://machinedb.systems.ethz.ch.

5.1 Message passing tree topologies

We evaluate the performance of Smelt's adaptive tree against the tree topologies shown in §2.4. We run atomic broadcasts, reductions, barriers and two-phase commit [23] as workloads on all of our machines.

We measure how long it takes until every thread has completed the execution of the collective operation. We avoid relying on synchronized clocks by introducing an additional message to signal completion: for atomic broadcast and two-phase commit a distinct leaf sends a message to the root. We measure the time until the root (i.e. the initiator of the operation) receives this message. We repeat this for all leaves and select the maximum time

machine	A MC 4x12x1	I IB 2x10x2	I NL 4x8x2	A BC 8x4x1	I KNC 1x61x4	I SB 2x8x2
CPU	AMD Opteron 6174	Intel Xeon E5-2670 v2	Intel Xeon L7555	AMD Opteron 8350	Xeon Phi	Intel Xeon E5-2660 0
micro arch	Magny Cours	IvyBridge	Nehalem	Barcelona	k1om	SandyBridge
#cores	4x12x1 @ 2.20GHz	2x10x2 @ 2.50GHz	4x8x2 @ 1.87GHz	8x4x1 @ 2.00GHz	1x60x4 @ 1.00GHz	2x8x2 @ 2.20GHz
caches	64K/ 512K/ 4M	32K/ 256K/ 25M	32K/ 256K/ 24M	64K/ 512K/ 2M	32K/ 512K/	32K/ 256K/ 20M
memory	126G	252G	110G	15G	15G	64G
machine	A SH 4x4x1	A IL 4x4x2	I SB 4x8x2	I BF 2x4x2	A IS 4x6x1	
CPU	AMD Opteron 8380	AMD Opteron 6378	Intel Xeon E5-4640 0	Intel Xeon L5520	AMD Opteron 8431	
micro arch	Shanghai	Interlagos	SandyBridge	Bloomfield	Istanbul	
#cores	4x4x1 @ 2.50GHz	4x4x2 @ 2.40GHz	4x8x2 @ 2.40GHz	2x4x2 @ 2.27GHz	4x6x1 @ 2.50GHz	
caches	64K/ 512K/ 6M	16K/2048K/ 6M	32K/ 256K/ 20M	32K/ 256K/ 8M	64K/ 512K/ 4M	
memory	16G	512G	505G	24G	15G	

Table 1: Machines used for evaluation. Caches given as L1 data / L2 / L3. (L2 per core, L3 socket). Number of cores represented as sockets / cores per socket / threads per core.

among them. In reductions, this is reversed as the root sends the message to the leaf. For barriers, we measure the cost on each core and take the maximum. Barriers are implemented as shown in §4.2.2. We repeat the experiment 10'000 times and collect 1'000 data points.

We first show a detailed breakdown for selected machines to motivate that no single tree topology performs well across all machines followed by giving an overview of the performance across all evaluated machines. There, we show that Smelt not only matches the best tree topology on each machine, but further improves performance on top of that.

Breakdown for selected machines Figure 8 shows the detailed comparison of an 4-socket AMD machine (A IL 4x4x2), a 4-socket Intel machine (I SB 4x8x2) and an Intel's Xeon Phi coprocessor (I KNC 1x60x4). The latter uses a ring topology to connect cores instead of a hierarchical interconnect. In our evaluation, we use only one thread per physical core.

Our results support the claim that there is no clear best static topology for all machines and that the best choice depends on the architecture and workload. As expected, sequential sending results in a significant slowdown compared to all other topologies. The other hardware oblivious trees, binary and Fibonacci, perform comparably but suffer from using too many inter-socket messages on hierarchical machines. The cluster topology performs well in many cases, but since it relies on the machine's hierarchy, it is slow on machines not having a hierarchical memory topology (I KNC 1x60x4). On A IL 4x4x2, the cluster is comparable but slower than Smelt. This is because of the rather static ordering for sending the messages as well as a node's outdegree in the tree that is not optimized. Despite using machine characteristics, the MST topology does not consider the protocol's communication patterns nor tries to maximize parallelism.

In contrast, Smelt's adaptive tree (AT) achieves good performance across all configurations due to the fact that it uses hardware information enriched with real measurements to capture fine-grained performance characteristics of the machine and adapt the message scheduling accordingly. Our results show that *generating* a tree based on our machine model as described in § 3.3 achieves good results without the programmer's awareness of hardware characteristics and manual tuning.

We demonstrate that the tree topology matters and that there is no static topology that performs best on all machines even when considering the NUMA hierarchy. We show that Smelt is able to adapt to a wide set of microarchitectures and machine configurations without manual tuning.

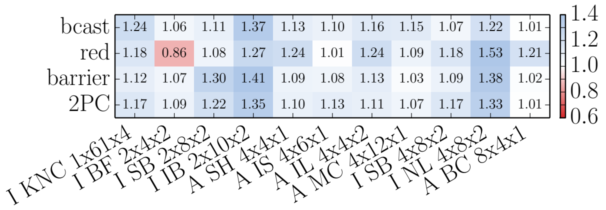


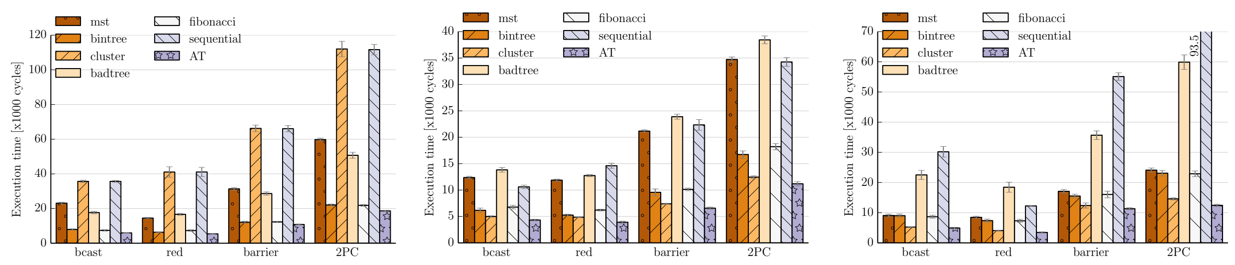
Figure 9: Comparison of Smelt to the best static tree topology on each machine. Ordered by the the number of sockets as indicated by the label.

Comparison with the best other topology. Figure 9 is a heat-map showing the speedup of Smelt compared to the best static tree on all machines. For example, if the "cluster" topology is the best tree topology besides Smelt on a machine, we use that as a baseline. All tree topologies use Smelt's transport layer (§4.1).

The Fibonacci tree achieves the best performance on three of these machines, the binary tree on one of them and the cluster tree on the remaining seven.

Smelt not only matches the best tree topology for all but one configuration, but also manages to achieve an average speedup of 1.16 over all machines, peaking at a speedup of up to 1.24x compared to the best static tree on AMD (A SH 4x4x1) and up to 1.53x on Intel (I NL 4x8x2).

To conclude, this experiment shows that even when the best static topology for a concrete machine is known, Smelt still manages to further improve the performance since, in addition to considering the hierarchy of the machine to avoid expensive cross-NUMA links, it optimally configures the outdegree in each node of the tree. It does



(a) I KNC 1x60x4 (b) A IL 4x4x2 (c) I SB 4x8x2

Figure 8: Details of micro benchmark results for all the evaluated tree topologies.

so based on the machine characteristics gathered from the pairwise send and receive measurements (§3.2).

5.2 Multicast topologies

Certain workloads require collective communications within a subset of the threads (i.e. multicast). We evaluate this scenario by running the benchmarks from § 5.1 with an increasing thread count starting from 2 up to the maximum number of threads of the machine. For this, we assign threads to NUMA nodes in round-robing, e.g. when showing four cores on a machine with four NUMA nodes, we would place one thread on each NUMA node.

Figure 10 shows the multicast scaling behavior of Smelt compared to the static trees on A IL 4x4x2 and A IS 4x6x1. For a low number of cores — for example one core per NUMA node to implement consistent updates to data replicated on a per-NUMA basis as in §5.7 — it is often best to send messages sequentially: we observe this behavior in both Figure 10b and 10a. At that point, all communication links are remote and the hierarchical cluster approach does not work well. When more cores are involved and the effects of local vs. remote communication become more obvious, the cluster topology performs best again. In summary, the choice of topology does not only matter on the machine, but also the multicast group intended to use.

Further, the performance benefit when using Smelt is often higher in intermediate configurations: for example in Figure 10b, the maximum speedup over the best static topology is 1.25 with 12 cores as to compared to 1.02 for a reduction involving all 24 cores. The maximum negative speedup of Smelt is 0.97 on A IS 4x6x1 for 22 cores.

5.3 Comparison with MPI and OpenMP

We compare Smelt with two established communication standards: MPI and OpenMP. MPI (Message Passing Interface) [30] is a widely used standard for message-based communication in the HPC community. MPI supports a wide range of collective operations, including broadcasts, reductions and barriers. Furthermore, the MPI libraries provide specific channels and optimizations for shared-memory systems.

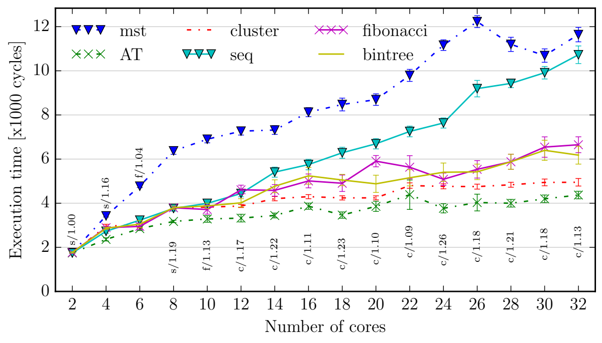
OpenMP 4.0 [39] is a standard for shared-memory parallel processing and is supported by major compilers, operating systems and programming languages. The OpenMP runtime library manages the execution of parallel constructs. Threads are implicitly synchronized after a parallel block or explicitly by the barrier directive.

We compare the collectives of MPI (Open MPI v1.10.2) and OpenMP (GOMP from GCC 4.9.2) with Smelt. For MPI we compare broadcasts, reductions, and barriers. OpenMP only provides reductions and barriers.

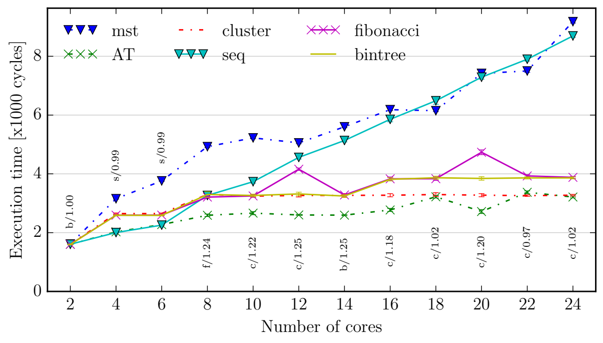
For each of the runtimes, we execute the experiment 3000 times and take the last 1000 measurements. At the beginning of each round, we synchronize all the threads with two dissemination barriers so that all threads enter the collective operation at the same time. This is different from the benchmark in § 5.1 since here the cost of the extra message depends on the used library. The broadcast is executed with a one byte payload and reduction has a single integer payload.

Figure 11 shows the results of the largest machines. Smelt outperforms MPI for all the tested collective operations. In broadcasts and reductions Smelt outperforms MPI with speedups between 1.6x and 2.6x. For barriers the lead is smaller (between 1.19x and 1.41x). OpenMP performs worse than MPI, showing that message-passing approaches are better-suited for large multicore machines than shared-memory programming models. Moreover, OpenMP is clearly outperformed by Smelt in reductions on the three machines. OpenMP barriers perform well on A IL 4x4x2 but still Smelt is between 1.5x and 3.8x faster.

Since OpenMP uses explicit and implicit barriers after each parallel construct, we extend the evaluation to demonstrate how Smelt can be used to improve the GOMP library [15]. GOMP's standard barriers are based on atomic instructions and the futex syscall on Linux [12, 14, 26]. We replaced GOMP's barrier with Smelt and compared it against the vanilla version. As workload we took *syncbench* and *arraybench* from the EPCC OpenMP micro-benchmarks suite [27] using standard settings and 5000 outer repetitions. We ran the benchmark using all

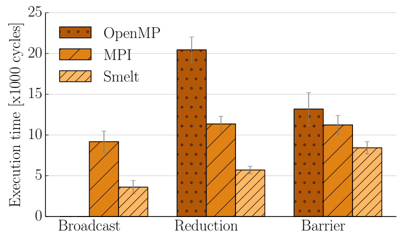


(a) Broadcast A IL 4x4x2

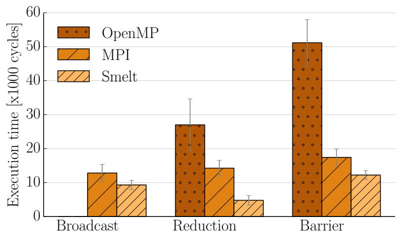


(b) Reduction A IS 4x6x1

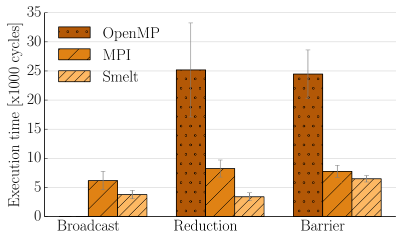
Figure 10: Multicast, cores allocated round-robin to NUMA nodes. Labels: speedup compared to the best static topology and first letter of that topology's name.



(a) A IL 4x4x2



(b) I NL 4x8x2



(c) I SB 4x8x2

Figure 11: Comparison with MPI and OpenMP.

available threads.

The results of the benchmark are shown in Table 2. Overall, Smelt performs significantly better or comparable to the original GOMP barriers. In the *BARRIER* micro-benchmark, we achieve up to 2.5x and 1.5x speedup respectively. These results show that replacing the standard barriers in GOMP with Smelt reduces the overhead for synchronization significantly.

5.4 Barriers micro-benchmarks

Barriers are important building blocks for thread synchronization in parallel programs. We compare our barrier implementation (§ 4.2.2) with the state-of-the-art MCS dissemination barrier [1] (parlibMCS) and a 1-way dissemination barrier that uses atomic flags [33] (dissemination). We show that our simple barrier implementation, based on broadcast and reduction, can compete with highly-tuned state-of-the-art shared-memory implementations.

In this evaluation, we synchronize threads using the different barriers in a tight for-loop of 10,000 iterations. This is yet another barrier benchmark and cannot be directly compared with the previous sections.

The results in Table 3 show significant differences between machines and whether or not hyperthreads are used. Whereas on A IL 4x4x2 Smelt performs worse and there is no clear winner, Smelt is up to 3x faster on Intel machines relative to the dissemination barrier and up to 6x faster compared to parlibMCS.

machine	C	parlibMCS		dissemination		Smelt	
I SB 4x8x2	32	16,718	(495)	6,699	(63)	4,725	(6)
	64	38,494	(755)	19,762	(22)	6,348	(10)
I NL 4x8x2	32	13,836	(348)	5,777	(239)	4,035	(22)
	64	15,604	(1,366)	6,333	(185)	5,755	(26)
A IL 4x4x2	16	4,288	(7)	4,596	(92)	4,792	(9)
	32	5,989	(23)	5,220	(12)	7,016	(35)

Table 3: Barrier micro-benchmark for 32 and 64 threads, median of 100 calls [cycles], standard error in brackets.

With this evaluation we have shown how a competitive barrier can be implemented easily using Smelt's hardware aware collective operations.

5.5 Streamcluster

PARSEC Streamcluster [6] solves the online clustering problem. We chose this benchmark because it is synchronization-intensive. We evaluate the performance of Smelt's barriers compared to PARSEC's default barriers, pthread barriers, and parLib dissemination barrier [1].

For all configurations, we used the native data set and run the benchmark with and without Shoal, a framework for optimizing memory placement and access based on access patterns [21]. Otherwise, Streamcluster's performance is limited by memory bandwidth and the effects of optimizing synchronization are less visible.

Our results in Table 4 confirm that optimizing both

		PARALLEL	FOR	BARRIER	SINGLE	COPYPRIV	COPY	PRIV
A IL 4x4x2	GOMP	21.81 (112.16)	6.93 (0.15)	6.92 (0.15)	11.31 (1.41)	287.84 (122.97)	101.40 (1.22)	52.20 (301.56)
	Smelt	13.93 (0.15)	4.15 (0.07)	4.13 (0.07)	7.83 (0.08)	120.10 (24.52)	104.66 (0.94)	13.84 (0.28)
I SB 4x8x2	GOMP	55.41 (333.78)	9.50 (3.22)	9.51 (3.23)	14.85 (2.26)	319.44 (7.29)	135.16 (1.13)	46.69 (140.40)
	Smelt	38.31 (0.17)	5.64 (0.02)	5.63 (0.02)	10.27 (0.98)	141.95 (1.81)	128.48 (1.04)	37.20 (0.44)

Table 2: EPCC OpenMP benchmark. Average in microseconds, standard error in brackets.

	pthread	parsec	Smelt	parlib MCS
A IL 4x4x2				
no Shoal	215.619	207.929	181.405	202.801
Shoal	51.816	52.075	41.116	41.061
I SB 4x8x2				
no Shoal	236.476	235.394	124.492	125.184
Shoal	66.421	68.079	28.283	28.779

Table 4: Execution time of Streamcluster [seconds]

memory accesses and synchronization primitives matter for achieving good performance of parallel programs. We also show that our simple barrier implementation (§4.2.2) based on a generic broadcast tree performs better than shared-memory barriers and is competitive with a state-of-the-art dissemination barrier.

5.6 Agreement

We implemented the 1Paxos [10] agreement protocol using Smelt. 1Paxos is a Paxos-variant optimized for multicore environments. Normal operation is shown in Figure 12a: a client sends a request to the leader which forwards the request to the acceptor. Then there is a single broadcast from the acceptor to the replicas that we optimize using Smelt. Upon receiving the broadcast, the leader responds to the client.

We re-implemented 1Paxos with and without Smelt, because the original 1Paxos paper uses its own threading and message passing library. Furthermore, they create one thread for each incoming connection which has a large negative impact on performance [19]. The processing time then dominates over communication cost making it unsuitable for the evaluation of our work.

We vary the number of replicas from 8 to 28 and use 4 cores as load generators, which was sufficient to issue enough requests to keep the system busy. The measurements are averaged over three runs of 20 seconds each. Figures 12b and 12c present the performance of the agreement protocol and an atomic broadcast with the same threads.

The results show that the agreement protocol on multicore machines can benefit from an optimized broadcast primitive: using Smelt improves the throughput and response time up to 3x compared to sequential sending on 28 replicas. As we increase the number of replicas, the sequential broadcast quickly becomes the bottleneck. Our results show that 1Paxos is highly tuned towards multicores as its scaling behavior and performance are similar to a plain broadcast.

By using Smelt, we can improve the performance of agreement protocols on multicore machines, improving also the scalability to larger number of replicas.

5.7 Key-value store

We implemented a replicated key-value store (KVS) using 1Paxos from § 5.6 to ensure consistency of updates while reads are served directly by the replica. In our implementation, the nearest replica responds to the client request. Our implementation supports a get/set interface. We focus on small keys (8 byte) and values (16 byte) to avoid fragmentation. If fragmentation was implemented, larger messages would simply be split up in multiple smaller messages. This would cause a behavior similar to adding more clients to the system.

We placed a KVS instance on each NUMA node of the machine, 8 on A IL 4x4x2. An increasing number of clients connect to their local KVS instance and issue requests. We executed the benchmark for 20 seconds and 3 runs with a get/set ration of 80/20.

The *set* performance results are shown in Figure 13. We omit the *get* results as they are served locally. Our results demonstrate that Smelt is able to improve performance even for a small number of replicas. Scalability and stability under high load are even better, resulting in up to 3x improvement for throughput and response time.

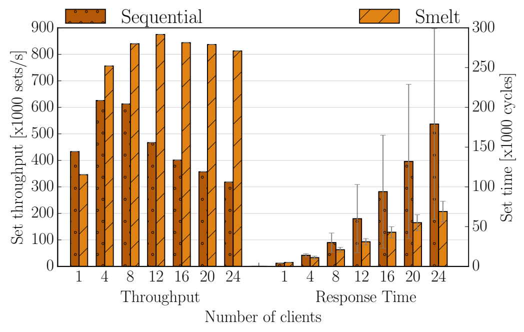
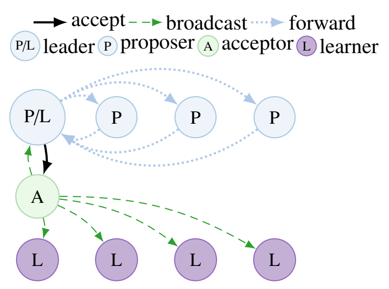


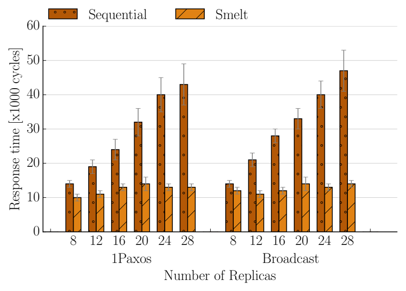
Figure 13: *Set* performance on A IL 4x4x2

6 Conclusion

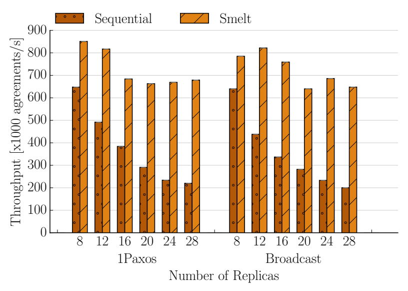
Smelt is a new approach for tuning broadcast algorithms to multicore machines. It automatically builds efficient broadcast topologies and message schedules tuned



(a) 1Paxos failure-free case.



(b) Response time on A IL 4x4x2.



(c) Throughput on A IL 4x4x2.

Figure 12: Performance results of 1Paxos agreement.

to specific hardware environments based on a machine model which encodes both static hardware information and costs for sending and receiving messages, generated from micro-benchmarks that capture low-level machine characteristics. Smelt provides an easy-to-use API which can be used to build high-level applications on top of it. We have shown that the trees generated by Smelt match or outperform the best static topology on each of a variety of machines.

Moreover, we also show how other collective operations can be constructed using these trees as a building block and achieve good performance without requiring any extra tuning effort. Barriers implemented trivially on top of Smelt outperform state-of-the-art techniques, including shared-memory algorithms which do not use message passing. We also achieve good scaling with the number of parallel requests in an in-memory replicated key-value store built on top of Smelt's adaptive trees.

Consequently, we claim that automatically generated broadcast topologies can deliver high performance in parallel applications without requiring programmers to have detailed understanding of a machine's topology or memory hierarchy. Smelt is open source and available for download at https://github.com/libsmelt.

6.1 Future work

We believe that using Smelt will have an even larger benefit on future machines that experience partial transient hardware failures [7, 13], are more heterogeneous [7] and increasingly rack-scale.

In the case of failures, the tree has to be reconfigured dynamically. To make this fast, the changes in the tree topology should ideally be kept local. We believe that our optimizations to the basic tree topology as described in §3.3.2 are a good starting point to explore strategies to efficiently update trees topologies locally. Furthermore, the asymmetry of group communication as a result of spacial scheduling and failures will make static tree topologies less ideal (e.g. the cluster works best for a "regular" symmetric machine and less for irregular topologies).

Smelt will likely be able to handle more heterogeneous hardware, as microbenchmarks reflect these performance characteristics and the adaptive tree automatically handles such cases. We show this by running Smelt on the Xeon Phi, which exposes a completely different architecture, without having to modify Smelt. Furthermore, simulation based on pairwise send and receive cost should work equally well on multicore machines providing message-passing hardware, as long as t_{send} and $t_{receive}$ adequately represent the system's cost for communication. However, it is likely that more microbenchmarks have to be added in the future to precisely capture hardware characteristics as more intricate accelerator hardware is added (e.g. deeper write buffers). However, the general approach of modeling the machine combined with simulation will likely still work in such a setting.

Further, we plan to extend our machine model to include more hardware details that may have an additional impact in larger machines, like contention on inter-socket links and write-buffer effects. Also, to date Smelt does not address partial failures, and assumes the entire machine to be fail-stop. Extending the system to support multiple failure domains and fully networked communication at rack scale is a natural line of extension of our work.

Acknowledgments

We thank our mentor Peter Chen and the anonymous reviewers for their detailed and useful reviews and the Computer Architecture Group from the University of A Coruña for the access to their cluster Pluton (Project TIN2013-42148-P).

References

[1] Amplab, UC Berkeley. PARLIB, MCS Locks. Online. http://klueska.github.io/parlib/mcs.html. Accessed 05/10/2016.

[2] A. Baumann, P. Barham, P.-E. Dagand, T. Harris, R. Isaacs, S. Peter, T. Roscoe, A. Schüpbach, and

A. Singhania. The Multikernel: A new OS Architecture for Scalable Multicore Systems. In *Proceedings of the 22nd ACM Symposium on Operating System Principles*, SOSP '09, pages 29–44, Big Sky, Montana, USA, 2009.

[3] A. Baumann, S. Peter, A. Schüpbach, A. Singhania, T. Roscoe, P. Barham, and R. Isaacs. Your computer is already a distributed system. Why isn't your OS? In *Proceedings of the 12th Workshop on Hot Topics in Operating Systems*, May 2009.

[4] S. Bell, B. Edwards, J. Amann, R. Conlin, K. Joyce, V. Leung, J. MacKay, M. Reif, L. Bao, J. Brown, M. Mattina, C.-C. Miao, C. Ramey, D. Wentzlaff, W. Anderson, E. Berger, N. Fairbanks, D. Khan, F. Montenegro, J. Stickney, and J. Zook. TILE64 - Processor: A 64-Core SoC with Mesh Interconnect. In *Digest of Technical Papers of the IEEE International Solid-State Circuits Conference*, ISSCC 2008, pages 88–598, Feb 2008.

[5] B. N. Bershad, T. E. Anderson, E. D. Lazowska, and H. M. Levy. User-level Interprocess Communication for Shared Memory Multiprocessors. *ACM Transactions on Computer Systems*, 9(2):175–198, May 1991.

[6] C. Bienia, S. Kumar, J. P. Singh, and K. Li. The PARSEC Benchmark Suite: Characterization and Architectural Implications. In *Proceedings of the 17th International Conference on Parallel Architectures and Compilation Techniques*, PACT '08, pages 72–81, Toronto, Ontario, Canada, 2008. ACM.

[7] S. Borkar and A. A. Chien. The future of microprocessors. *Communications of the ACM*, 54(5):67–77, May 2011.

[8] J. Bruck, R. Cypher, and C.-T. Ho. Multiple Message Broadcasting with Generalized Fibonacci Trees. In *Proceedings of the 4th IEEE Symposium on Parallel and Distributed Processing*, pages 424–431, Arlington, Texas, USA, Dec 1992.

[9] T. David, R. Guerraoui, and V. Trigonakis. Everything You Always Wanted to Know About Synchronization but Were Afraid to Ask. In *Proceedings of the 24th ACM Symposium on Operating Systems Principles*, SOSP '13, pages 33–48, Farminton, Pennsylvania,USA, 2013. ACM.

[10] T. David, R. Guerraoui, and M. Yabandeh. Consensus Inside. In *Proceedings of the 15th International Middleware Conference*, Middleware '14, pages 145–156, Bordeaux, France, 2014. ACM.

[11] N. Dershowitz and S. Zaks. Enumerations of Ordered Trees. *Discrete Mathematics*, 31(1):9–28, 1980.

[12] U. Drepper. Futexes Are Tricky. Technical report, Red Hat, Inc., Dec 2011.

[13] P. Faraboschi, K. Keeton, T. Marsland, and D. Milojicic. Beyond Processor-centric Operating Systems. In *Proceedings of the 15th USENIX Conference on Hot Topics in perating Systems*, HOTOS'15, pages 17–17, Kartause Ittingen, Switzerland, 2015. USENIX Association.

[14] H. Franke and R. Russell. Fuss, Futexes and Furwocks: Fast Userlevel Lockingin Linux. In *Proceedings of the 2002 Ottawa Linux Symposium*, OLS '02, pages 18:1–18:11, Denver, Colorado, USA, 2002.

[15] Free Software Foundation, Inc. Welcome to the home of GOMP. Online. https://gcc.gnu.org/projects/gomp/. Accessed 05/10/2016.

[16] J. Giacomoni, T. Moseley, and M. Vachharajani. FastForward for Efficient Pipeline Parallelism: A Cache-optimized Concurrent Lock-free Queue. In *Proceedings of the 13th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '08, pages 43–52, Salt Lake City, Utah, USA, 2008. ACM.

[17] R. L. Graham and G. Shipman. MPI Support for Multi-core Architectures: Optimized Shared Memory Collectives. In *Proceedings of the 15th European PVM/MPI Users' Group Meeting*, EuroPVM/MPI ' 08, pages 130–140, Dublin, Ireland, 2008. Springer Science & Business Media.

[18] D. Hackenberg, D. Molka, and W. E. Nagel. Comparing Cache Architectures and Coherency Protocols on x86-64 Multicore SMP Systems. In *Proceedings of the 42nd Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO 42, pages 413–422, New York, New York, USA, 2009. ACM.

[19] R. Haecki. *Consensus on a Mulicore Machine*. ETH Zurich, 2015. Master's Thesis, http://dx.doi.org/10.3929/ethz-a-010608378.

[20] Hewlett-Packard, Intel, Microsoft, Phoenix, Toshiba. *Advanced Configuration and Power Interface Specification, Rev. 4.0a*, Apr. 2010. http://www.acpi.info/.

[21] S. Kaestle, R. Achermann, T. Roscoe, and T. Harris. Shoal: Smart Allocation and Replication of Memory for Parallel Programs. In *Proceedings of the 2015 USENIX Annual Technical Conference*, USENIX ATC '15, pages 263–276, Santa Clara, CA, 2015.

[22] D. E. Knuth. *The Art of Computer Programming*, volume 3. Addison-Wesley, 2nd edition, 1998.

[23] B. W. Lampson and H. E. Sturgis. Crash Recovery in a Distributed Data Storage System, 1979.

[24] B. Lepers, V. Quema, and A. Fedorova. Thread and memory placement on numa systems: Asymmetry matters. In *Proceedings of the 2015 USENIX Annual Technical Conference*, USENIX ATC '15, pages 277–289, Santa Clara, CA, July 2015.

[25] S. Li, T. Hoefler, and M. Snir. NUMA-aware Shared-memory Collective Communication for MPI. In *Proceedings of the 22nd International Symposium on High-performance Parallel and Distributed Computing*, HPDC '13, pages 85–96, New York, New York, USA, 2013. ACM.

[26] Linux Programmer's Manual. futex - fast user-space locking. Online. http://man7.org/linux/man-pages/man2/futex.2.html. Accessed 05/10/2016.

[27] Mark Bull and Fiona Reid. EPCC OpenMP micro-benchmark suite. Online. https://www.epcc.ed.ac.uk/research/computing/performance-characterisation-and-benchmarking/epcc-openmp-micro-benchmark-suite. Accessed 05/10/2016.

[28] D. Molka, D. Hackenberg, and R. Schöne. Main Memory and Cache Performance of Intel Sandy Bridge and AMD Bulldozer. In *Proceedings of the Workshop on Memory Systems Performance and Correctness*, MSPC '14, pages 4:1–4:10, Edinburgh, United Kingdom, 2014. ACM.

[29] D. Molka, D. Hackenberg, R. Schöne, and W. E. Nagel. Cache Coherence Protocol and Memory Performance of the Intel Haswell-EP Architecture. In *Proceedings of the 44th International Conference on Parallel Processing*, ICPP ' 15, pages 739–748, Beijing, China, 2015.

[30] MPI Forum. Message Passing Interface Forum. Online. Accessed 05/10/2016.

[31] R. C. Prim. Shortest connection networks and some generalizations. *The Bell System Technical Journal*, 36(6):1389–1401, Nov 1957.

[32] S. Ramos and T. Hoefler. Cache Line Aware Optimizations for ccNUMA Systems. In *Proceedings of the 24th International Symposium on High-Performance Parallel and Distributed Computing*, HPDC '15, pages 85–88, Portland, Oregon, USA, 2015. ACM.

[33] S. Ramos and T. Hoefler. Cache Line Aware Algorithm Design for Cache-Coherent Architectures. *IEEE Transactions on Parallel and Distributed Systems (TPDS)*, 27(10):2824–2837, Oct 2016.

[34] RRZE - Regionales RechenZentrum Erlangen. likwid. Online, 2015. https://github.com/RRZE-HPC/likwid.

[35] T.-I. Salomie, I. E. Subasu, J. Giceva, and G. Alonso. Database Engines on Multicores, Why Parallelize when You Can Distribute? In *Proceedings of the 6th Conference on Computer Systems*, EuroSys '11, pages 17–30, Salzburg, Austria, 2011. ACM.

[36] Silicon Graphics International Corporation. libnuma. Online, 2015. http://oss.sgi.com/projects/libnuma/.

[37] P. J. Slater, E. J. Cockayne, and S. T. Hedetniemi. Information dissemination in trees. *SIAM Journal on Computing*, 10(4):692–701, 1981.

[38] Y.-H. Su, C.-C. Lin, and D. Lee. Broadcasting in Heterogeneous Tree Networks. In *Proceedings of the 16th Annual International Conference on Computing and Combinatorics*, pages 368–377. Springer-Verlag, 2010.

[39] The OpenMP Architecture Review Board. The OpenMP API specification for parallel programming. Online. http://openmp.org/. Accessed 05/10/2016.

Light-weight Contexts: An OS Abstraction for Safety and Performance

James Litton[1,2], Anjo Vahldiek-Oberwagner[2], Eslam Elnikety[2], Deepak Garg[2], Bobby Bhattacharjee[1], and Peter Druschel[2]

[1]University of Maryland, College Park
[2]Max Planck Institute for Software Systems (MPI-SWS), Saarland Informatics Campus

Abstract

We introduce a new OS abstraction—light-weight contexts (*lwC*s)—that provides independent units of protection, privilege, and execution state within a process. A process may include several *lwC*s, each with possibly different views of memory, file descriptors, and access capabilities. *lwC*s can be used to efficiently implement roll-back (process can return to a prior recorded state), isolated address spaces (*lwC*s within the process may have different views of memory, e.g., isolating sensitive data from network-facing components or isolating different user sessions), and privilege separation (in-process reference monitors can arbitrate and control access).

*lwC*s can be implemented efficiently: the overhead of a *lwC* is proportional to the amount of memory exclusive to the *lwC*; switching *lwC*s is quicker than switching kernel threads within the same process. We describe the *lwC* abstraction and API, and an implementation of *lwC*s within the FreeBSD 11.0 kernel. Finally, we present an evaluation of common usage patterns, including fast roll-back, session isolation, sensitive data isolation, and in-process reference monitoring, using Apache, nginx, PHP, and OpenSSL.

1 Introduction

Processes abstract the unit of isolation, privilege, and execution state in general-purpose operating systems. Computations that require memory isolation, privilege separation, or continuations at the OS level must be run in separate processes[1]. Unfortunately, switching and communicating between processes incurs the cost of invoking the kernel scheduler, resource accounting, context-switching, and IPC. The actual hardware-imposed cost of isolation and privilege separation, however, is much smaller: if the TLB is tagged with an address space identifier, then switching context requires as little as a system call and loading a CPU register.

Just as threads separate the unit of execution from a process, we assert that there is benefit to decoupling memory isolation, execution state, and privilege separation from processes. We show that it is possible to isolate memory and privileges, and maintain multiple execution states *within* a process with low overhead, thus streamlining common computation patterns and enabling more efficient and safe code.

[1]Language runtimes can provide these properties at the expense of additional overhead, language dependence, and an increased trusted computing base.

We introduce a new first-class OS abstraction: the *light-weight context* (*lwC*). A process may contain multiple *lwC*s, each with their own virtual memory mappings, file descriptor bindings, and credentials. Optionally and selectively, *lwC*s may share virtual memory regions, file descriptors and credentials.

*lwC*s are not schedulable entities: they are completely orthogonal to threads that may execute within a process. Thus, a thread may start in *lwC a*, and then invoke a system call to *switch* to *lwC b*. Such a switch atomically changes the VM mappings, file table entries, permissions, instruction and stack pointers of the thread. Indeed multiple threads may execute simultaneously within the same *lwC*. *lwC*s maintain per-thread state to ensure a thread that enters a *lwC* resumes at the point where it was created or last switched out of the *lwC*.

*lwC*s enable a range of new in-process capabilities, including fast roll-back, protection rings (by credential restriction), session isolation, and protected compartments (using VM and resource mappings). These can be used to implement efficient in-process reference monitors to check security invariants, to isolate components of an app that deal with encryption keys or other private information, or to efficiently roll back the process state.

We have implemented *lwC*s within the FreeBSD 11.0 kernel. The prototype shows that it is possible to implement *lwC*s in a production OS efficiently. Our experience with implementing and retrofitting large applications such as Apache and nginx with *lwC*s has taught us that it is possible to introduce many new capabilities, such as rollback and secure data compartments, to existing production code with minimal overhead. This paper's contributions are:

- We introduce *lwC*s, a first-class OS abstraction that extends the POSIX API, and present common coding patterns demonstrating its different uses.

- We describe an implementation of *lwC*s within FreeBSD, and show how *lwC*s can be used to implement efficient session isolation in production web servers, both process-oriented (Apache, via roll-back) and event-driven (nginx, via memory isolation). We show how efficient snapshotting can provide session isolation while improving performance on web-based applications using a PHP-based MVC application on nginx. We show how cryptographic libraries such as OpenSSL can efficiently create isolated data compartments *within a process* to render sensitive data (such as private keys) immune to external attacks (e.g., buffer overruns a la Heartbleed [7]). Finally, we show how *lwC*s can efficiently implement in-process reference monitors, again for industrial-scale servers such as Apache and nginx, that can introspect on system calls and memory.

• We evaluate *lwC*s using a range of micro-benchmarks and application scenarios. Our results show that existing methods for session isolation are often slower than *lwC*s. Other common uses such as *lwC*-supported sensitive data compartments and reference monitoring have little to negligible overhead on production servers. Finally, we show that using the *lwC* snapshot capability to quickly launch an initialized PHP runtime can improve the performance of a production server.

The rest of this paper is organized as follows: we discuss related work in Section 2 and describe the *lwC* abstraction, API, and design in Section 3. We present common *lwC* coding patterns in Section 4. We describe our FreeBSD implementation of *lwC*s in Section 5, and present an experimental evaluation in Section 6. We conclude in Section 7.

2 Related work

Wedge [5] provides privilege separation and isolation among *sthreads*, which otherwise share an address space. Sthreads are implemented using Linux processes. *lwC*s are orthogonal to threads and therefore avoid the cost of scheduling when switching contexts. Moreover, *lwC*s can snapshot and resume an execution in any state, while a sthread can only revert to its initial state. Wedge provides a software analysis tool that helps refactor existing applications into multiple isolated compartments. *lwC*s could benefit from a such a tool as well.

Shreds [9] builds on architectural support for *memory domains* in ARM CPUs, a compiler toolchain, and kernel support to provide isolated compartments of code and data within a process. Like *lwC*s, shreds provide isolated contexts within a process. *lwC*s, however, are fully independent of threads, require no compiler support, and rely on page-based hardware protection only. *lwC*s also provide protection rings and snapshots, which shreds do not.

In SpaceJMP [12], address spaces are first-class objects separate from processes. While both systems can switch address spaces within a process, SpaceJMP and *lwC*s provide different abstractions, capabilities, and are motivated by entirely different applications. SpaceJMP supports applications that wish to use memory larger than the available virtual address bits allow, wish to maintain pointer-based data structures beyond process lifetime, and share large memory objects among processes. A SpaceJMP context switch is not associated with a mandatory control transfer and, therefore, SpaceJMP does not support applications that require isolation or privilege separation within a process. *lwC*s, on the other hand, provide in-process isolated contexts, privilege separation, and snapshots.

Dune [4] provides a kernel module and API that export the Intel VT-x architectural virtualization support safely to Linux processes. Privilege separation, reference monitors, and isolated compartments can be implemented within a process using Dune. *lwC*s instead provide a unified abstraction and API for these capabilities, and their implementation does not rely on virtualization hardware, the use of which could interfere with execution on a virtualized platform. While the *lwC* implementation incurs a higher cost for system call redirection, it avoids Dune's overhead on TLB misses and kernel calls.

In Trellis [20], code annotations, a compiler, run time, and OS kernel module provide privilege separation within an application. The kernel and runtime ensure that functions can be called and data accessed only by code with the same or higher privilege level. *lwC*s provide privilege separation without language/compiler support, and can switch domains at lower cost. Moreover, *lwC*s additionally support snapshots.

Switching among *lwC*s is similar to migrating threads in Mach [13], where they were implemented to optimize local RPCs. Migration of threads across address spaces is also an element of the model described by Lindström et al. [18] and the COMPOSITE OS [24]. In single address space operating systems (SASOS) like Opal [8] and Mungi [15], all processes as well as persistent storage share a single large (64-bit) address space. Unlike *lwC*s, these systems do not provide privilege separation, isolation, or snapshots *within* a process.

Mondrian Memory Protection (MMP) [32] and Mondrix [33] use hardware extensions to provide protection at fine granularity within processes. The CHERI [31, 34] architecture, compiler, and operating system provides hybrid hardware-software object capabilities for fine-grained compartmentalization within a process. *lwC*s provide in-process isolation at page granularity without specialized hardware or language support.

Resource containers [3] separate the unit of resource accounting from a process and account for all resources associated with an application activity, even if the activity requires processing in multiple processes and the kernel. *lwC*s are orthogonal to resource containers.

The Corey [6] OS provides fine-grained control over the sharing of memory regions and kernel resources among CPU cores to minimize contention. *lwC*s provide the orthogonal capability of in-process isolation, privilege separation, and snapshots.

Light-weight isolation, privilege separation, and snapshots can be provided also within a programming language. Functional languages like Scheme and ML provide closures through the primitive call/cc, which can be used to record a program state and revert to it later, and to implement co-routines. Typed object-oriented languages like C++ and Java provide *static* isolation and privilege separation through private and protected class fields but do not isolate objects of the same class from each other. *Dynamic* language-based protection, often implemented as object capabilities [14, 22, 23], provides fine-grained isolation and privilege separation but has considerable runtime overhead. *lwC*s instead provide in-process isolation, privilege separation, and snapshots at the OS level, independent of a programming language.

In low-level languages like C, isolation and privilege separation can be attained using binary rewriting and compiler-inserted checks as in CFI [1], CPI [17] and secure compilation [25]. All these techniques rely on dynamic checks that have runtime overhead. Techniques such as CPI and secure compilation rely on OS support for the isolation of a reference monitor, which *lwC*s can provide at low cost.

Software fault isolation (SFI) [29] and NaCl [35] rely on static checking and instrumentation of binaries to isolate memory within applications running on unmodified operating systems. SFI and NaCl do not selectively protect system calls and file descriptors. *lwC*s instead allow fine-grained control over memory, file descriptors and other process credentials, and provide snapshots as part of an OS abstraction.

Process checkpoint facilities create a linearized snapshot of a process's state [10,19,26,38]. The snapshot can be stored persistently and subsequently used to reconstitute the process and resume its execution on the same or a different machine. Checkpoint facilities are used for fault-tolerance and load balancing. *lwC*s instead provide very fast in-memory snapshots of a process's state.

The Determinator OS [2] relies on a private workspace model for concurrency control, which enables deterministic execution on multi-core platforms. To support the model, Determinator provides *spaces*, in which programs mutate private copies of shared objects. Like *lwC*s, spaces provide isolated address spaces. Unlike a *lwC*, however, a space is tied to one thread, does not have access to I/O or shared memory, and can interact only with its parent and only in limited ways.

Intel's Software Guard Extensions (SGX) [16] provide ISA support to isolate code and data in *enclaves* within a process. By mapping contexts to enclaves, SGX could be used to harden *lwC*s against a stronger threat model (untrusted OS) and to provide hardware attestation of contexts. However, enclaves have no access to OS services, so some *lwC* applications would need considerable rearchitecting to run on SGX.

NOVA [27] provides protection domains (separate address spaces) and execution contexts (an abstraction similar to threads) in a micro hypervisor. NOVA's goal is to isolate VMMs and VMs from the core hypervisor, which is different from *lwC*'s goal of providing isolation, privilege separation, and snapshots within processes.

3 *lwC* design

*lwC*s are separate units of isolation, privilege, and execution state within a process. Each *lwC* has its own virtual address space, set of page mappings, file descriptor bindings, and credentials. Threads and *lwC*s are independent. Within a process, a thread executes within one *lwC* at a time and can switch between *lwC*s. *lwC*s are named using file descriptors. Each process starts with one *root* *lwC*, which has a well-known file descriptor number.

Table 1 shows the *lwC* API. A *lwC* may create a new (child) *lwC* using the `lwCreate` operation and receive the child's file descriptor. If a context *a* has a valid descriptor for *lwC c*, a thread executing inside *a* may switch to *c* using the `lwSwitch` operation. A *lwC c* is terminated (and its resources released) when the last *lwC* with a descriptor for *c* closes the descriptor. Common usage patterns of the *lwC* API will be shown in Section 4.

3.1 Creating *lwC*s

The `lwCreate` call creates a new (child) *lwC* in the current process. The operation's default semantics are similar to that of a POSIX *fork*, in that the child *lwC*'s initial state is an identical copy of the calling (parent) *lwC*'s state, except for its descriptor. Unlike with fork, however, child and parent *lwC* share the same process id, and no new thread is created. No execution takes place in the new *lwC* until an existing thread switches to it.

`lwCreate` returns the descriptor of the new child *lwC* *new* to the parent *lwC* with the *caller* descriptor set to -1. When a thread switches to the new *lwC* (*new*) for the first time, the `lwCreate` call returns with the caller's *lwC* descriptor in *caller* and the parent's *lwC* descriptor in *new*, along with any arguments from the caller in args.

By default, the new *lwC* gets a private copy of the calling *lwC*'s state at the time of the call, including per-thread register values, virtual memory, file descriptors, and credentials. Shared memory regions in the calling *lwC* are shared with the new *lwC*. The parent *lwC* may modify the visibility of its resources to the child *lwC* using the resource-spec argument, described in Section 3.3.

The implementation does not stop other threads executing in the parent *lwC* during an `lwCreate`. To ensure that the child *lwC* reflects a consistent snapshot of the parent's state, all threads that are active in the parent at the time of the `lwCreate` therefore should be in a consis-

Function	Return Value		System Call
Create *lwC*	{*new*, *caller*, args}	←	`lwCreate`(resource-spec, options)
Switch to *lwC*	{*caller*, args}	←	`lwSwitch`(*target*, args)
Resource access	status	←	`lwRestrict`(*l*, resource-spec)
	status	←	`lwOverlay`(*l*, resource-spec)
	status	←	`lwSyscall`(*target*, mask, syscall, syscall-args)

Table 1: API for interacting with *lwC*s. Parameters in italics *new, caller, …* are *lwC* descriptors. Arguments args are passed during *lwC* switches; resource-spec denotes resources (e.g. memory pages, file descriptors) that can be shared or narrowed.

tent state. The application may achieve this, for instance, by barrier synchronizing such threads with the thread that calls `lwCreate`. A thread that does not exist in the parent *lwC* at the time of the `lwCreate` may not switch to the child in the future.

The `lwCreate` call takes several option flags. `LWC_SHAREDSIGNALS` controls signal handling in the child *lwC*, as described in Section 3.7. `LWC_SYSTRAP` indicates that any system calls for which the child does not hold the required OS capability should be redirected to its parent. This feature enables a parent to interpose and mediate its child's system call activity, as described in Section 3.6.

The fork semantics of `lwCreate` enable the convenient, language independent creation of *lwC*s based on the current state of the calling *lwC*. No additional APIs are required to initialize a new *lwC*. The new *lwC* can be viewed also as a snapshot of the state of the caller at the time of invoking `lwCreate`, enabling the caller to revert to this state in the future.

3.2 Switching between *lwC*s

The `lwSwitch` operation switches the calling thread to the *lwC* with descriptor *target*, passing args as parameters. `lwSwitch` retains the state of the calling thread in the present *lwC*. When this *lwC* is later switched back into by the same thread, the call returns with the switching *lwC* available as *caller* and arguments passed in args.

Note that returns from a `lwSwitch` and `lwCreate`, any signal handlers that were installed, and the instruction pointer locations of threads in a parent *lwC* at the time of a `lwCreate` define the only possible entry points into a *lwC*. (The root *lwC* has an additional one-time entry point when the process is launched.)

`lwSwitch` is semantically equivalent to a coroutine `yield`. In fact, as far as control transfer is concerned, *lwC*s can be viewed as isolated and privilege separated coroutines. Recall that a procedure is a special case of a coroutine. To achieve a (remote) procedure call among *lwC*s, the called procedure, when done, simply switches to its caller and then loops back to its beginning. This functionality can be provided easily as part of a library.

3.3 Static resource sharing

When a *lwC* is created using `lwCreate`, the child *lwC* receives a copy-on-write snapshot of all its parent's resources by default. The parent can modify this behavior using the *resource-spec* argument in the `lwCreate` operation. The *resource-spec* is an array of C unions: each array element specifies either a range of file descriptors, virtual memory addresses, or credentials. For each range, one of the following sharing options can be specified. `LWC_COW`: the child receives a logical copy of the range of resource (the default). `LWC_SHARED`: the range of resources is shared among parent and child. `LWC_UNMAP`: the range of resources is not mapped from the parent into the child. (The child may subsequently map different resources in the address range.)

When restricting the resources inherited by the child, care must be taken to minimally pass on the stacks, code, synchronization variables, and other dependencies of all threads in the parent *lwC*, to ensure predictable behavior if these threads switch to the child in the future.

3.4 Dynamic resource sharing

A *lwC* may dynamically map (overlay) resources from another *lwC* into its address space using the `lwOverlay` operation. The caller specifies which regions of a given resource type (file descriptor or memory) are to be overlayed, and whether the specified region should be copied or shared, in the *resource-spec* parameter. The `lwOverlay` call will only succeed if the caller *lwC* holds *access capabilities* (described below in Section 3.5) for the requested resources. A successful `lwOverlay` operation unmaps any existing resources at the affected addresses in the caller's address space.

3.5 Access capabilities

Access capabilities are associated with *lwC* file descriptors. Each *lwC* holds a descriptor with a universal access capability for itself. When a *lwC* is created, its parent receives a descriptor with a universal access capability for the child. A parent *lwC* may grant a child *lwC* access capabilities for the parent *lwC* selectively by marking resource ranges as `LWC_MAY_ACCESS` in the *resource-spec* argument passed to the `lwCreate` call.

Access capabilities may be restricted on a *lwC* descriptor with the `lwRestrict` call. The *resource-spec* parameter restricts the set of resources that may be overlayed or accessed by any context that holds the *lwC* descriptor *l*. The valid resource types are file descriptors, virtual memory addresses, and syscall numbers. Subsequent to the call, the descriptor will allow `lwOverlay` to succeed for any file descriptors and memory addresses, and `lwSyscall` for any syscalls, respectively, that are within the intersection of the resource-spec set and whatever capabilities *l* had previous to the call.

3.6 System call interposition/emulation

Consider an *lwC* *C* that was created with the `LWC_SYSTRAP` flag. If a thread in *C* invokes a system call for which *C* does not hold a capability according to the OS's sandboxing mechanism, the thread is switched to its parent *lwC* instead, if the thread exists in the parent (if the thread does not exist in the parent, the call fails with an error). When the thread is resumed in the parent *lwC* as a result of a faulting syscall by the child, the arguments in the switch contain the system call number attempted and the arguments passed to it. The parent can choose to decline the syscall and return an error to the child, or perform a syscall on behalf of the child, possibly with different arguments (see below). To signal the completion of the child's system call, the thread executing in the parent *lwC* switches back to the child with the return value and any error code as arguments to the switch call.

An authorized *lwC* may perform a syscall on behalf of another *lwC target* using the `lwSyscall` operation. The `lwSyscall` succeeds if the *lwC* calling the operation holds an access capability (see Section 3.5) for the *target* and syscall, and holds the OS credentials required to perform the requested syscall. The effects of a successful execution of `lwSyscall` are as if the *target* had executed the requested syscall, except that it returns to the calling context. The mask parameter allows the caller to modify this behavior by specifying aspects of its own context that are to be put in place for the duration of the system call. Specifically, the caller may specify that the target's file table, memory space, credentials, or any combination be replaced by the caller's equivalent for the duration of the call. This allows the efficient implementation of useful patterns, such as enabling a untrusted *lwC* to read (or append) a fixed number of bytes from (to) a protected file *without* having access to the file descriptor.

3.7 Signal handling

*lwC*s modify the standard POSIX signal handling semantics in the following way. We distinguish between *attributable* signals, which can be attributed to the execution of a particular instruction in a *lwC*, and *non-attributable* signals, which cannot. Attributable signals, such as `SIGSEGV` or `SIGFPE`, are delivered to the *lwC* that caused the signal immediately. Non-attributable signals, such as `SIGKILL` or `SIGUSR1`, are delivered to the root *lwC* and any *lwC*s in the process that were created with the LWC_SHARESIGNALS option by a parent *lwC* that is able to receive such signals. A non-attributable signal is delivered to a *lwC* upon the next switch to the *lwC*.

3.8 System call semantics

*lwC*s modify the behavior of some existing POSIX system calls. During a `fork`, all *lwC*s in the calling process are duplicated in the child process. Any memory regions that were `mmap`'ed as MAP_SHARED in some *lwC*s of the calling process are shared with the corresponding *lwC*s in the new child process, *within and across* the two processes. Any memory regions that are shared among *lwC*s in the parent process using the LWC_SHARED option in `lwCreate` are shared among the corresponding *lwC*s *within* the child process only. An *exit* system call in any *lwC* of a process terminates the entire process.

3.9 *lwC* isolation

Because *lwC*s do not have access to the state of each others' memory, file descriptors, and capabilities unless explicitly shared, they can provide strong isolation and privilege separation within a process. Since *lwC*s share executable threads, however, an application needs to make certain assumptions about the behavior of other *lwC*s in the same process, even if they don't share resources and don't have overlay capabilities for each other. Specifically, a *lwC* can block or execute a thread indefinitely or terminate the process prematurely by invoking `exit`.

We believe these assumptions are reasonable in practice because the *lwC*s of a process are part of the same application program. Denial-of-service within a process is self-defeating. On the other hand, *lwC*s can reliably prevent accidental leakage of private information across user sessions, isolate authentication credentials and other secrets, and ensure the integrity of a reference monitor.

A *lwC* can learn about certain activities of other *lwC*s by registering for non-attributable signals. An application that wishes to limit information flow across *lwC*s should create *lwC*s without the LWC_SHARESIGNALS option (the default).

3.10 *lwC* security

*lwC*s provide isolation and privilege separation within a process, but include powerful mechanisms for sharing and control among the *lwC*s of a process. Therefore, it is important to understand the threat model and the security properties provided by the *lwC* abstraction.

Threat model We assume that the kernel is trustworthy and uncompromised, and that the tool chain used to build, link, and load the application does not have exploitable vulnerabilities that can be used to hijack control before main() starts. When a *lwC* is created, its parent has universal privileges on the *lwC*. Consequently, the security of a *lwC* assumes that its parent (and, by transitivity, all its ancestors) cannot be hijacked to abuse these privileges. In practice, the parent should drop all unnecessary privileges on the child immediately after the child is created, so this assumption is needed only with respect to the remaining privileges. When an application uses dynamic sharing, the same assumption must be extended to all *lwC*s that obtain privileges indirectly. The *lwC* API does not enable any inter-process communication or sharing beyond the standard POSIX API. Consequently, no new assumptions regarding *lwC*s in other processes are needed.

Security properties The properties of a *lwC* are constrained by the properties of the process in which it exists. A *lwC* cannot attain privileges that exceed those of its process, and the confidentiality and integrity properties of any *lwC* cannot be weaker than those of its process. The properties of the root *lwC* are those of the process. In applications that do not use dynamic sharing, the privileges of a non-root *lwC* are bounded by those of its parent and, transitively, by those of its ancestors; its integrity and confidentiality cannot be weaker than those of any of its ancestors. In applications that use dynamic sharing through the exchange of access capabilities via a common ancestor, the integrity (confidentiality) of a *lwC* depends on all siblings and descendants that have write (read) rights to it. For this reason, dynamic sharing should be used with caution.

In typical patterns of privilege separation, the root *lwC* should run a high-assurance component, i.e., one that is simple, heavily scrutinized, and exports a narrow interface. A component that protects sensitive state is at or near the root, to minimize its dependencies. More complex, less stable, network or user-facing components should be encapsulated in de-privileged *lwC*s at the leaves of a process's *lwC* tree and should execute with the least privileges required.

4 Common *lwC* usage patterns

In this section, we illustrate *lwC* use patterns for snapshots, isolation and protection rings. For some of the patterns, we use a web server as an illustrative setting. However, all the patterns are broadly applicable.

Snapshot and rollback A common *lwC* use pattern is snapshot and rollback, where a service process (such as a server worker process) initializes its state to the point where it is ready to serve requests (or sessions), snapshots this state, serves a request and rolls its state back to the snapshot before serving the next request. As compared to a setup where the process manually cleans up request-specific state after each request, the snapshot and rollback can improve performance by efficiently discarding the request-specific state with a single call, and also improves security by isolating *sequential* requests served by the same task from each other.

Algorithm 1 shows the pseudocode of a small library containing two functions—snapshot() and rollback()—and a main() server function illustrating their use. The server initializes its state and calls snapshot() on line 12 to create a snapshot. snapshot() duplicates the current *lwC* (copy-on-write) using `lwCreate` on line 2. The descriptor of the duplicated snapshot, called new, is returned at line 4 and stored in the variable snap. The program serves the request and then, to reset its state, calls rollback(). Control transfers to line 2 *in the snap* (the child) and then immediately to line 6 where the original *lwC* is closed (its resources are reclaimed). The snap recursively calls snapshot() (line 7). At line 2, it creates a duplicate of itself and returns that duplicate to main() at line 12. The cycle then repeats, with snap and its duplicate having taken the roles of the original *lwC* and the snap, respectively.

Algorithm 1 Snapshot and rollback

```
 1: function SNAPSHOT()
 2:     new,caller,arg = lwCreate(default_spec, ... )
 3:     if caller = -1 then                          ▷ parent
 4:         return new
 5:     else
 6:         close(caller)
 7:         return snapshot()
 8: function ROLLBACK(snap)                          ▷ never returns
 9:     lwSwitch(snap, 0)
10: function MAIN()
11:     ...                                          ▷ initialize state
12:     snap = snapshot()
13:     ...                                          ▷ serve request
14:     rollback(snap)
            ▷ kills current lwC, continues at line 12 in snap
```

In our evaluation, we use this pattern to roll back the state of pre-forked worker processes after each session in the Apache web server.

Isolating sessions in an event-driven server High throughput servers like nginx handle several sessions in single-threaded processes using event-driven multiplexing. However, they provide no isolation among sessions within a process. This shortcoming can be addressed using *lwC*s. Algorithm 2 illustrates the usage pattern.

The program defines a set of network socket descriptors to poll, one for each client connection, on line 10

Algorithm 2 Event-driven server with session isolation

```
 1: function SERVE_REQUEST(retlwc, client)
 2:     loop
 3:         if would_block(client) then
 4:             lwSwitch(retlwc, 0);
 5:         else if finished(client) then
 6:             lwSwitch(retlwc, 1);
 7:         else
 8:             serve(client)
 9: function MAIN
10:     descriptors = { accept_ descriptor }
11:     file2lwc_map = { accept_descriptor => root }
12:     loop
13:         next = descriptors.ready()
14:         if next = accept_descriptor then
15:             fd = accept(next)
16:             descriptors.insert(fd)
17:             specs = { ... }          ▷ Share fd descriptor only
18:             new,caller,arg = lwCreate(specs, ...)
19:             if caller = -1 then               ▷ context created
20:                 file2lwc_map[fd] = new
21:             else
22:                 serve_request(root, fd)
23:         else
24:             lwc = file2lwc_map[next]
25:             from, done = lwSwitch(lwc, ...)
26:             if done = 1 then
27:                 close(next);close(from)
28:                 descriptors.remove(next)
29:                 file2lwc_map.unset(next)
```

and sets a mapping of the listening socket descriptor to the current *lwC* on line 11.

Once a descriptor is ready the program moves past line 13 and either accepts and encapsulates a new descriptor in a worker *lwC* or resumes execution of a previous one that is now ready. In the former case, the worker's *lwC* is created on line 18 such that no descriptor other than `fd` is passed to it (line 17), the created *lwC* descriptor is mapped on line 20 and the loop resumes. In the latter case, the previously mapped worker *lwC* is retrieved on line 24. This *lwC* is now immediately switched into on the subsequent line. At this point execution resumes on line 18 *in the worker*. As a result, it enters the `serve_request` function on line 22.

When the worker is done executing it switches back into the root *lwC*. It uses the `lwSwitch` argument to indicate whether it is done with its work (arg = 1) or not (arg = 0). When it switches back to the root, control flow resumes at line 25. Depending on the argument passed in from the worker, the root *lwC* either closes the socket and the worker or leaves them intact for later service.

Since all worker *lwC*s obtain a private copy of the root's state, no worker sees session-specific state of other workers. This isolates the sessions from each other.

Sensitive data isolation A third common use pattern isolates sensitive data within a process by limiting access to a single *lwC* that exposes only a narrow interface. As an illustration, Algorithm 3 shows how to isolate a private signature key that is available to a signing function, but kept hidden from the rest of the (large and network-facing) program.

Algorithm 3 Sensitive Data Isolation

```
 1: function SIGN(key, data, out_buffer)
 2: function SIGN_SSTUB(caller,arg)
 3:     loop
 4:         lwOverlay(caller,{VM,arg,sizeof(arg),SHARE})
 5:         sign(privkey, arg.in, arg.out)
 6:         lwOverlay(caller,{VM,arg,sizeof(arg),UNMAP})
 7:         caller,arg = lwSwitch(caller, 0)
 8: function SIGN_CSTUB(buf)
 9:     caller,res = lwSwitch(child, buf)
10: function MAIN
11:     ...                         ▷ initialization, load privkey
12:     child,caller,arg =
13:     lwCreate({VM,0,MAX,MAY_OVERLAY}, 0)
14:     if caller != -1 then
15:         sign_sstub(caller,arg)
16:     privkey = 0                               ▷ erase key
17:     lwRestrict(child, {VM,0,MAX,NO_ACCESS})
18:     loop
19:         ...
20:         sign_cstub(buf)
21:         ...
```

The main function initializes the program and loads the private signing key into the variable `privkey` (line 11). Next, it calls `lwCreate` to create a second *lwC* with the same initial state (line 13). The child *lwC*, which will become the isolated compartment with access to the `privkey`, is granted the privilege to overlay any part of the parent's virtual memory.

The parent *lwC* continues executing on line 16, where it deletes its copy of the private signing key and then revokes its privilege to overlay any part of the child *lwC*'s memory. Any code executed in the parent after this point (line 17) has no way to access the private key. When this code wishes to sign data, it calls `SIGN_CSTUB` passing as argument a structure that contains the data to sign and a large enough buffer to hold the returned signature.

The `SIGN_CSTUB` function performs a `lwSwitch` to the child *lwC*, passing a pointer to the buffer as the argument. The first time the child is switched to, it returns from `lwCreate` with caller != -1 and calls `SIGN_SSTUB` (line 15), from which it does not return.

SIGN_SSTUB now uses lwOverlay to map the buffer from the parent *lwC* as a shared region into its own address space (line 4), calls the SIGN function with the private key, and then unmaps the buffer from its address space. Finally, the function calls lwSwitch to return control to the parent *lwC*, which resumes by returning from the lwSwitch in line 9. Upon future invocations of SIGN_CSTUB, the child *lwC* returns from the lwSwitch in line 7 and loops back.

In our evaluation with web servers, we use this pattern to isolate parts of the OpenSSL library that handle long-term private keys, thus protecting the keys from vulnerabilities like the widespread Heartbleed bug [7]. (Heartbleed remains a threat even after global key revocations and reissues [11, 37].)

Protected reference monitor Next, we describe a pattern that allows a parent *lwC* to intercept any subset of system calls made by its child and monitor those calls. In our evaluation, we use this pattern to implement a reference monitor for system calls made by the web server.

Algorithm 4 Reference Monitor

```
 1: function MONITOR(child)
 2:     _,call = lwSwitch(child, NULL)
 3:     loop
 4:         if is_allowed(call) then
 5:             spec = { type = CRED, SANDBOX }
 6:             rv = lwSyscall(child, spec,
                                 call.num, call.params)
 7:             out.err,out.rv = errno, rv;
 8:         else
 9:             out.err,out.rv = EPERM, -1;
10:         _,call = lwSwitch(child, out)
11: function MAIN
12:     specs = { ... } ▷ Share (COW) all but private data
13:     child,c,_ = lwCreate(specs, LWC_SYSTRAP)
14:     if c = -1 then                 ▷ parent becomes refmon
15:         monitor(child)                      ▷ Never returns
16:     privdrop() && run()                 ▷ Child starts here
```

Algorithm 4 shows the pseudocode of the pattern for the case where the monitoring parent is the root *lwC*. On line 13, the root creates a child *lwC* but reserves a private region, which may contain secrets (e.g., encryption keys) of which the child is not allowed to get a copy. The child is created with the flag LWC_SYSTRAP, so any system calls that the child lacks the capability for trap to the root *lwC*. Once the child *lwC* is created, the root *lwC* enters the monitoring function, which never returns.

Within the monitoring function, the root, now acting as the reference monitor, yields to the child immediately (line 2). The reference monitor regains control when the child makes a system call that it does not have the capabilities for. The reference monitor checks whether the call should be allowed (line 4) and, if so, makes the call *in the context of the child* (line 6). It yields to the child with the system call's result and error code. If the system call should be disallowed, the reference monitor yields to the child with error code EPERM. The reference monitor loops to handle the next system call.

The child starts execution on line 16 where it immediately drops privileges for all system calls that should be monitored. This causes all these system calls to trap to the reference monitor, which handles them as described above.

For simplicity, our example reference monitor merely filters system calls, a capability already provided by many operating systems. A more interesting monitor could inspect the system call arguments or other parts of the child's state by overlaying in the appropriate regions, or perform arbitrary actions and system calls on behalf of the child.

5 Implementation

We have implemented *lwC*s in the FreeBSD 11.0. We begin with a brief background of the FreeBSD kernel structures used in implementing *lwC*s.

5.1 FreeBSD Background

In implementing *lwC*s, we had to modify FreeBSD kernel data structures corresponding to process memory, file tables and credentials.

Memory In FreeBSD, the address space of a process is organized under a vmspace structure (described fully in [21]). Within the address space, there are virtual memory regions that correspond to a contiguous interval of memory mapped into the process's virtual address space. These memory regions are represented as vm_map_entry structures. Attempting to access any memory that is not within a memory region results in a segmentation fault.

Two memory regions that are contiguous and have the same protection bits can be merged into a single vm_map_entry. The number of memory regions within a process is typically small (few tens), though for some processes (notably Apache, that maps modules into different regions) it can be larger. Work performed during fork and lwCreate is proportional to the number of vm_map_entry structures.

Switching the virtual address space map of a process during a context switch (*lwC* or otherwise) can be a relatively efficient operation on modern processors. Previous generations of processors required a TLB flush whenever the address space had to be changed, as is the case during process context switches, or *lwC* switches. Modern processors include a "process context identifier" (PCID) that can be used to distinguish pages that belong to different page tables. (On current Intel processors, the PCID is 12-bits, enabling 4096 different page tables to be distinguished.) TLB entries are tagged with the PCID that was active when they were resolved. Whenever the active page table is ready to be changed, the kernel sets the CR3 register to a value containing the PCID and the address of the first page directory entry. Any cached TLB entries that share this PCID are considered valid and may be used. Importantly, the entire TLB does not have to be flushed upon a context switch since entries belonging to other PCIDs are simply considered invalid by the hardware. This facility reduces the cost of context switches by reducing the frequency of TLB flushes. FreeBSD 11.0 supports PCIDs and each *lwC* is assigned a unique one for every core it is activated on.

File Table In FreeBSD, all files, sockets, devices, etc. open in a process are accessible via the process's file table, which is held as a reference in the process structure. Each entry contains a cursor, per-process flags, and access capabilities. In our implementation, *lwC*s are also accessed via file-table entries. Upon `fork`, the file table is copied from the parent to the child process.

Credentials Process credentials determine capabilities and privileges, and include process user identifiers (uid, gid), limits (cpu time, maximum number of file descriptors, stack size, etc.), the current FreeBSD jail (a restrictive chroot-like environment) the process is operating in, and other accounting information.

The credentials of a process are attached to the process structure via a `struct ucred` pointer. Upon a `fork`, a reference to the parent structure is given to the child; system calls that modify the credential structure allocate a new `struct ucred` for the process, and copy unmodified fields from the parent.

5.2 *lwC* Implementation

Like a process, each *lwC* has a file table, virtual memory space, and credentials associated with it.

Memory Unless otherwise specified, `lwCreate` replicates the `vmspace` associated with the parent *lwC* in exactly the same manner as `fork`. However, any memory regions that are specified as `LWC_UNMAP` during the `lwCreate` call are not mapped into the new *lwC*'s address space. Any memory regions that are marked as `LWC_SHARE` are mapped into the *lwC* as memory that differs from shared memory in only one respect: a subsequent `fork` will not share this region with its parent. During a `lwSwitch`, the calling thread saves its CPU registers, releases its reference to the current vmspace structure, and acquires a reference from the address space of the switched to *lwC*.

File Table By default, during a call to `lwCreate` all file descriptors are copied into the *lwC* file table in the same manner as `fork` except that any associated file descriptor overlay rights are copied as well, as described in section 5.2. If the user specifies an interval in the resource specifier as `LWC_UNMAP`, the corresponding descriptors are not copied into the file table. The user may specify that the entire file table is to be shared; in this scenario, as an optimization, we store a reference to the parent *lwC*'s file table.

***lwC* descriptors** With one exception, *lwC* descriptors have the same visibility as regular file descriptors. Upon `lwCreate`, if the file table or a *lwC* descriptor is not shared, then the child *lwC* is not able to access the parent's *lwC*s. *lwC*s closed with the `close` syscall results in their removal from the calling *lwC*'s file table. Upon a `lwCreate` or `lwSwitch`, if a *caller* parameter is specified, then the newly created (or switched to) *lwC a* inherits a reference to the caller *lwC b* as a file descriptor. This descriptor, corresponding to *b*, is inserted into *a*'s file table when *a* is switched to next. (If *a*'s file table already had a descriptor for *b*, then that descriptor is reused, and *a*'s file table is not modified.)

Credentials We copy credentials the same way that they are copied during a `fork` call. Restoring previous credentials (using a *lwC* switch) may reverse calls that dropped privileges/put the process into a sandbox. Our reference monitor example (Section 4) shows how this mechanism can be used. Credentials are treated similarly to file descriptors and `vmspace` structures. The calling thread's credential structure is replaced with a reference to the target *lwC*'s reference structure.

Permissions and Overlays An executing *lwC* interacts with another *lwC* within a process by either switching to it or by overlaying (some of) that *lwC*'s resources.

A *lwC a* may switch to a *lwC b* only if *b*'s descriptor is present in *a*'s file table. Overlay permissions are more fine-grained: upon creating a new *lwC c*, the parent *p* passes a set of resource specifiers. Some of these may have `LWC_MAY_OVERLAY` flag set, which allows *c* to overlay specified resources from *p*.

The `lwCreate` call (*p* creating *c*) results in two file descriptors. One refers to *c* and has full overlay rights, and is inserted into *p*'s file table. Thus the creator (parent) *lwC* obtains all rights to the child.

The second descriptor, given to *c*, refers to the *p lwC* and only allows overlays on the descriptor as specified by *p* in the `lwCreate` call. File descriptors duplicated via the `dup` or similar calls create a new descriptor with a copy of the overlay rights. These rights can be narrowed using the `lwRestrict` call.

The `lwOverlay` call imports resources from one *lwC* into the calling *lwC*, assuming permissions are not violated. File table entries that are masked by an overlay are closed prior to inserting new entries. Similarly, memory region overlays unmap existing regions in the calling *lwC* that are within the overlay interval prior to importing overlaid regions. If the `LWC_SHARE` flag is set, the memory will be shared with the target *lwC* (i.e., writes will be visible to both *lwC*s). This sharing does not persist past a `fork`.

Multi-Threaded Support Our implementation supports *lwC*s in multithreaded programs. In addition to necessary synchronization, *lwC*-specific state that used to be associated with a process (and shared amongst all threads) must instead be associated with each *lwC*. This does not affect the existing semantics of processes because in normal operation each thread has a reference counted pointer to shared objects (e.g., memory spaces). Once *lwC* system calls are invoked it is possible for two threads to reference separate address spaces (i.e., *lwC*s). The modifications to the existing kernel were largely superficial outside of process creation and destruction.

6 Evaluation

In this section, we evaluate *lwC*s using microbenchmarks, and when applying the usage patterns discussed in Section 4 in the context of the Apache and nginx web servers. Our experiments were performed on Dell R410 servers, each with 2x Intel Xeon X5650 2.66 GHz 6 core CPUs with both hyperthreading and SpeedStep disabled, 48GB main memory, running FreeBSD 11.0 (amd64) and OpenSSL 1.0.2. The servers were connected via Cisco Nexus 7018 switches with 1Gbit Ethernet links. Each server has a 1TB Seagate ST31000424SS disk formatted under UFS.

6.1 *lwC* switch

Table 2 compares the time to execute a `lwSwitch` call compared to context switching between processes (using a semaphore), between kernel threads (using a semaphore, which we found to be faster than a mutex), and user threads. The user threads use the `getcontext` and `setcontext` calls specified by POSIX.1-2001. A *lwC* switch takes less than half the time of a process or kernel thread switch. The reason is that a *lwC* switch avoids the synchronization and scheduling required for a process or thread context switch, instead requiring only a switch of the vm mapping. Somewhat surprisingly, a kernel thread switch is on par with a process context switch when both use the same form of synchronization. The reason is that the kernel code executed during a switch between two kernel threads in the same process or in different processes is largely the same.

User threads are only moderately faster than *lwC* switches, because in FreeBSD 11, the user context switch is implemented by a system call. In Linux glibc, it is instead implemented in userspace assembly. In an experiment with Linux 3.11.10 on the same hardware, user thread switches run in 6% of the time required by semaphore-based kernel thread switches.

lwC	process	k-thread	u-thread
2.01 (0.03)	4.25 (0.86)	4.12 (0.98)	1.71 (0.06)

Table 2: Median switch time (in microseconds) and standard deviation over ten trials.

6.2 *lwC* creation

Next, we measured the total cost of creating, switching to, and destroying *lwC*s with default arguments (all resources shared COW with the parent) within a single process. When no pages are written in either the parent or child *lwC* during the lifetime of the child, the system is able to create, switch into once, and destroy an *lwC* in 87.7 microseconds on average, with standard deviation below 1%. This result is independent of the amount of memory allocated to the process. Each page written in either parent or child, however, causes a COW fault, which requires a page frame allocation and copy. When 100, 1000, 10000, and 100000 pages are written in the child during the experiment described above, the average total time taken per *lwC* increases to 397, 3054, 35563, and 34182 microseconds, respectively. Standard deviation was below 7% in all cases. The cost of maintaining a separate *lwC* is approximately linearly dependent on the number of unique pages it creates, and is lowest when *lwC*s in a process share most of their pages.

The results of our microbenchmarks can be used to estimate the cost of using *lwC*s in an application, given an estimate of the rate of *lwC* creations and switches, and the number of unique pages in each *lwC*. Later in this section, we evaluate the overhead of *lwC*s in the context of specific applications: Apache and nginx.

6.3 Reference monitoring

Following the pattern described in Section 4, we have implemented an in-process reference monitor using *lwC*s. When a process starts, the reference monitor gains control first and creates a child *lwC*, which executes the server application. The child *lwC* is sandboxed using FreeBSD Capsicum and disallowed from using certain system calls, which are instead redirected to the parent *lwC* using the LWC_SYSTRAP option. Our reference monitor restricts access to the filesystem, though other policies that restrict any system call or inspect memory (using `lwOverlay`) can readily be implemented within our basic schema. We compare the *lwC* reference monitor (**lwc-mon**) to two other techniques:

Inline Monitoring (inline) This is a baseline scheme where the reference monitor checks are inlined with the application code. The monitored process is

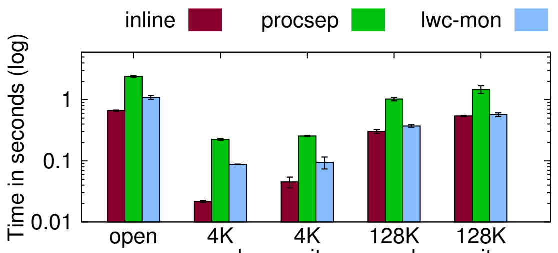


Figure 1: Cost of 10,000 monitored system calls in seconds (log scale). Error bars show standard deviation.

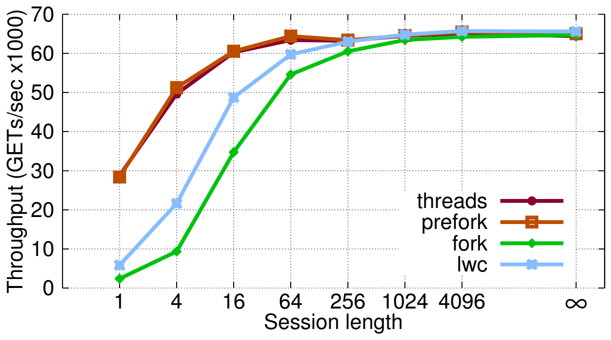
LD_PRELOADed with a library that intercepts each system call and checks arguments. Inlining provides a lower bound on overhead, but does not provide security since the monitored process can overwrite the checks or otherwise bypass the interception library.

Process Separation (procsep) This method provides a secure reference monitor in a separate process. The monitored process runs in a sandbox based on FreeBSD Capsicum [30]: the sandbox ensures that the monitored process is unable to issue prohibited system calls (e.g. **open**). At initialization, but prior to entering the sandbox, the monitored process connects to the reference monitor process over a Unix domain socket, which it can subsequently use to communicate with the reference monitor, even while sandboxed. All **open** calls (which the sandbox restricts) must be vectored through this socket, which allows the reference monitor to inspect and restrict the access as necessary. If the access is to be granted to the sandboxed application, the reference monitor shares a file descriptor over the socket.

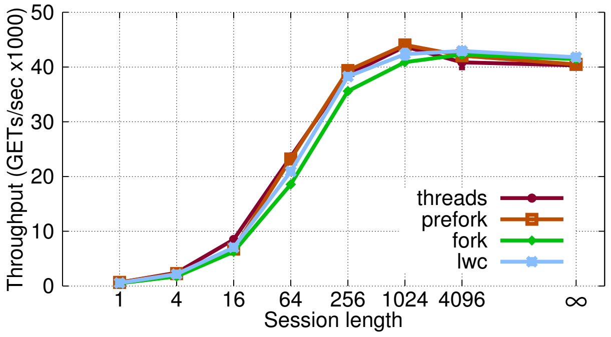
Figure 1 shows the overhead of monitoring open, read and write system calls, while an application is accessing a file stored in an in-memory file system. The application calls each system call 10,000 times and we report the average of 5 runs. Faster system calls have higher relative overhead since the fixed cost of redirecting the system call has to be paid. **lwc-mon** does not require data copying or IPC and hence outperforms **procsep** by a factor of two or more.

6.4 Apache

Modern web servers are designed to efficiently map user sessions to available processing cores. For instance, the popular Apache HTTP server provides multi-threading using kernel threads (**threads**) in one configuration and pre-forked processes that map to different cores (**prefork**) in another. Higher performance servers, such as nginx, use an event loop (based on kqueue or epoll) within a process, and have the option of spawning multiple processes that map to cores, each with their own event loop.



(a) HTTP



(b) HTTPS

Figure 2: Apache throughput in (GETs/sec) of 128 concurrent clients, 45 byte docs. Error bars show standard deviation, which was below 3.7%.

Consider the problem of isolating individual user sessions to separate the privileges of different user sessions or to implement per-user information flow control. None of the above mentioned server configurations provide such isolation: multi-threaded and event-driven configurations serve different sessions concurrently in the same process; pre-forked processes sequentially share among different sessions. Apache can be configured to fork a new process for each user session (**fork**), which provides memory isolation and privilege separation. As our results demonstrate, however, this configuration has low performance for small session lengths, due to the overhead of forking processes[2].

*lwC*s can provide memory isolation, privilege separation, and high performance. We have augmented the prefork mode in Apache (version 2.4.18) to provide session isolation using the snapshot and rollback pattern from Section 4. Within each Apache process, we create a *lwC* that serves a user session; when the session ends, the

[2]In fact, we had to patch Apache (in server/mpm_common.c) to continuously check the status of child processes (rather than at 1s intervals) to get this configuration to perform at all at small to modest session lengths.

lwC switches (reverts) to its initial (untainted) state before serving the next user session, thereby ensuring the isolation property.

In the following set of experiments, we use ApacheBench (ab) to issue HTTP and HTTPS requests to our Apache server. We modified ab to support varying client session lengths by using HTTP Keepalive and terminating a session after a certain number of requests. We launch a single ApacheBench instance which repeatedly makes up to 128 concurrent requests for a small 45 byte document. We chose small document requests to make sure the results are not I/O-bound. Figure 2 shows the number of GET requests served per second by the different Apache configurations at different session lengths, and for HTTP and HTTPS. For HTTPS, the server uses TLSv1.2, ECDHE-RSA-AES256-GCM-SHA384 with 4096 bit keys. The results were averaged over five runs of 60 seconds each.

At session length ∞, each client maintains a session for the duration of the experiment. The **threads** and **prefork** configurations, which provide no isolation, perform comparably for all session lengths and protocols. **fork** and **lwc** configurations provide isolation: **lwc** has better throughput in all cases, and has a significant advantage for short sessions (256 and below), particularly for HTTP. (In HTTPS, the high CPU overhead for session establishment dominates overall cost; however, emerging hardware support for crypto will diminish these costs, exposing once again the costs of isolation.) Moreover, **lwc** achieves performance comparable to the best configuration *without isolation* for sessions lengths of 256 and larger.

We also repeated the experiment with GET requests for 900 byte documents. These documents are 20x larger but still small enough not to saturate the network link. The trends and relative throughput between the different configuration were very close to those in Figure 2, with the absolute peak throughput within 10%.

We have integrated reference monitoring within Apache (and nginx). Figure 3 shows the throughput of Apache **prefork** in different reference monitor configurations when used to serve short (45 byte) documents. The results were averaged over five runs of 20 seconds each. In this experiment, the **open** and **stat** system calls are monitored and checked against a whitelist of allowed directories. These results show that a reference monitor implementation based on in-process *lwC* incurs lower overhead than an implementation based on process separation even for large applications where the monitored system calls constitute only part of what the applications do. The overhead of reference monitoring increases with session length due to the increase in relative number of reference monitored system calls (open and stat) compared to other system calls (accept, read, send, close).

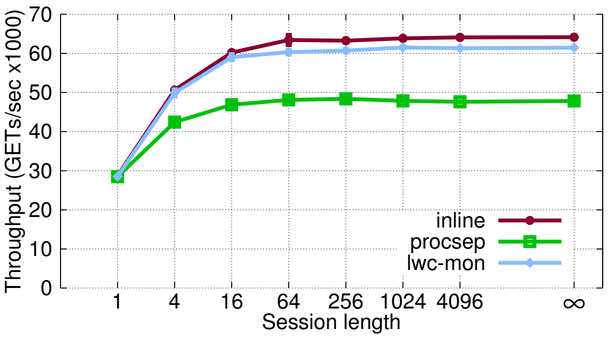


Figure 3: Throughput of different Apache reference monitoring configurations in (GETs/sec) of 128 concurrent clients, 45 byte docs. Error bars show standard deviation, which was below 2%.

6.5 Nginx

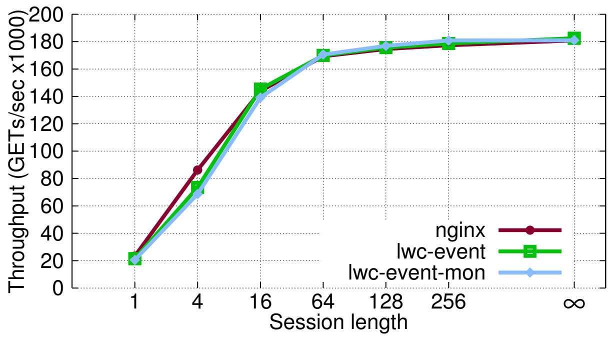
To enable session isolation in nginx (version 1.9.15), we allocate a *lwC* for each new connection: each event for a single connection is isolated within the *lwC*, following the session isolation pattern from Section 4. Note that in the nginx case, each process may serve many different connections simultaneously, and our implementation creates a *lwC* per active connection within the process. We have also integrated a reference monitor with nginx.

We experiment with different nginx configurations: the stock **nginx**, **lwc-event** augments nginx's event loop to create a new *lwC* per connection, and **lwc-event-mon** combines a reference monitor with the per-connection *lwC*. In each case we configured nginx to use 10 worker processes, as we found that this had the best performance. We launch four ApacheBench instances, each of which repeatedly makes up to 75 concurrent requests for a small 45 byte document.

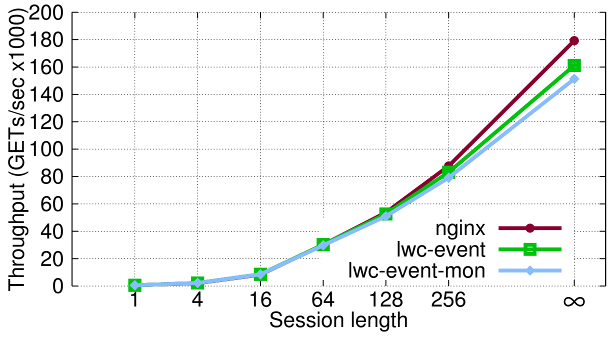
Figure 4 shows the average number of queries served by each of the configurations over five runs of 60 seconds each. The standard deviation did not exceed 0.9%.

nginx is considered the state of the art high-performance server. It uses a highly optimized event loop and is about 2.88x quicker than Apache. Introducing *lwC*s in this base configuration (named **lwc-event** in the results) has no significant impact on the throughput of this high-performance configuration. Similarly, reference monitoring adds only minimal overhead. For both HTTP and HTTPS, with isolation and reference monitoring, *lwC*-augmented nginx performs comparably to native nginx.

Large scale servers may need to maintain tens of thousands of concurrent user sessions. Using *lwC*s for session isolation increases the amount of per-session state. Therefore, our next experiment explores how using *lwC*s for session isolation affects nginx's performance under a



(a) HTTP



(b) HTTPS

Figure 4: Nginx throughput in GETs/sec with 10 workers, 45B documents, 300 concurrent requests. Error bars show standard deviation, which was below 0.9%.

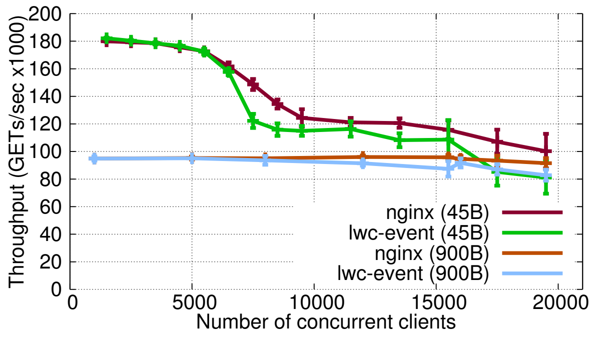


Figure 5: Nginx cumulative throughput in GETs/sec with 10 workers, session length 256, 45B and 900B documents, increasing number of concurrent clients. Error bars show standard deviation.

large number of concurrent client connections. We experimented with two configurations: in the first, we use between 6 and 76 ApacheBench instances, and each instance issues 250 concurrent requests for a 45 byte document. The session length was 256 and we used 10 nginx workers. The second configuration is identical except the ApacheBench instances request 900 byte documents.

Figure 5 shows the average number of requests served, over 5 runs of the experiment, as a function of the number of client sessions for stock nginx and **lwc-event**for both file sizes.

For small documents, **lwc-event** matches the performance of native nginx up to 6500 clients. Beyond, the performance of both configurations declines following the same trend, but the absolute throughput of **lwc-event** falls below that of nginx by up to 19% at 19,500 concurrent clients. In investigating this result further, we find that FreeBSD kernel threads, in particular, the interrupt handler thread, gets CPU bound after 6500 clients, and the CPU consumption of the nginx worker threads *reduces* with higher numbers of clients as the nginx worker threads block waiting for the kernel to demultiplex packets. The **lwc-event** configuration further pays an extra cost of *lwC* switches, which reduces performance compared to stock nginx. However, given that **lwc-event** provides session isolation, this is a still a strong result.

For 900 byte documents, the performance of stock nginx and **lwc-event** remain similar until ~12000 simultaneous clients. Performance of stock nginx is not affected by increasing numbers of clients: this is because the rate of incoming requests is lower, which means the kernel threads do not saturate the CPU. With increasing numbers of clients, eventually the cost of *lwC* switches, which were amortized over serving a larger document, become a measurable factor.

Overall, our results show that using *lwC*s, it is possible to implement features such as session isolation and reference monitoring at low cost for both HTTPS and HTTP sessions, and even in a high-performance server under a challenging workload.

6.6 Isolating OpenSSL keys

*lwC*s provide a particularly effective way to isolate sensitive data from network-based attacks such as buffer overflows or overreads. The sensitive data is stored in a *lwC*, within the process, such that the network-facing code has no visibility into pages that store the sensitive data. In this way, unless the kernel is compromised, the data is guaranteed safe, but access to functions that require the data can be rapid, using a safe *lwC*-crossing interface.

As an example, we have isolated parts of the OpenSSL library that manipulate secret information within Apache and nginx. In our case, the web server certificate private keys are isolated; note that such a scheme would have rendered attacks such as Heartbleed completely ineffective since the buffer overread that Heartbleed relied on would not have visibility into the memory storing the private keys. We evaluate this scheme using the following configurations:

In-process LwC Sensitive data is stored in a *lwC* within the process, following the pattern from Algorithm 3 in Section 4. The network-facing code within the process has no visibility into the sensitive data; access is through a narrow interface exported via *lwC* switch entry points. The isolated *lwC* has a copy of the original process at the time of creation and may call whatever functions are available within its address space. Our encapsulated OpenSSL library takes advantage of this fact because the isolated *lwC* hosts a COW copy of the OpenSSL code and global state and need not be aware that it is running in a restricted environment. None of the changes in the sensitive *lwC* are visible to the network facing code.

We evaluate the cost of providing this isolation by performing SSL handshakes (TLSv1.2,ECDHE-RSA-AES256-GCM-SHA384 with 4096 bit keys) with the nginx web server. The server was configured to spawn four worker processes. We used ApacheBench with concurrency level 24 and a session length of 1. In our experiments, native nginx required 99.7 seconds to complete ten thousand SSL handshakes, whereas the configuration with a *lwC* isolated SSL library required 100.4 seconds. With *lwC*s, isolating SSL private keys is essentially free.

Our prototype isolates only the server certificate private key, but not session keys or other sensitive information. More fine-grained isolation of the OpenSSL state, such as that described in [5], can be implemented readily using *lwC*s.

6.7 FCGI fast launch

We demonstrate the utility of *lwC* snapshotting by adding a "fast launch" capability to a PHP application. When a PHP request is served, a PHP script is read from disk, compiled by the interpreter, and then executed. During execution, other PHP files may be included and executed. We modified the PHP 7.0.11 programming language to add a `pagecache` call that allows the script to "fast-forward" using previous snapshots. Our implementation augments PHP-FPM [28], which functions as a FCGI server for nginx. Our test application is based on the MVC skeleton application that is included with the Zend PHP framework [36], which provides the core functionality for creating database-backed web-based applications such as blogs.

Before a PHP script performs any computation that depends on request-specific parameters (e.g., cookie information), the script may invoke the `pagecache` call, which implements the snapshot pattern (Algorithm 1). The first time a `pagecache` is invoked, we take a snapshot and then revert to it on subsequent requests to the same URL, effectively jumping execution forward in time. We use a shared memory segment to store data that must survive a snapshot rollback, including request-specific data and network connection information.

Our experiments run PHP-FPM with 11 workers. PHP itself includes an opcode cache (which caches the compilation of each script in memory) and our results include configurations where the PHP opcode cache is enabled and not. When combining the opcode cache and the *lwC* snapshot, we warm up the opcode cache before taking the snapshot. The results in Table 3 are an average of five runs and overall standard deviation was less than 2%.

stock php no cache	*lwC* php no cache	stock php cache	*lwC* php cache
226.1	615.8	1287.5	1701.4

Table 3: Average requests per second over 60 seconds with 24 concurrent requests.

With or without the opcode cache, the *lwC* snapshot is able to skip over much of the initialization of the runtime and whatever PHP execution would otherwise occur before the `pagecache` call. This result is remarkable in that it shows *lwC*s can provide significant performance benefit to highly optimized end-to-end applications such as web frameworks, *while adding isolation between user requests.*

7 Conclusions

We have introduced and evaluated light-weight contexts (*lwC*s), a new first-class OS abstraction that provides units of isolation, privilege, and execution state independent of processes and threads. *lwC*s provide isolation and privilege separation among program components within a process, as well as fast OS-level snapshots and coroutine style control transfer among contexts, with a single abstraction that naturally extends the familiar POSIX API. Our results show that fast roll-back of FCGI runtimes, compartmentalization of crypto secrets, isolation and monitoring of user sessions can be implemented in the production Apache and nginx web server platforms with performance close to or better than the original configurations in most cases.

8 Acknowledgments

We would like to thank the anonymous reviewers, Paarijaat Aditya, Björn Brandenburg, Mike Hicks, Pete Keleher, Matthew Lentz, Dave Levin, Neil Spring, and our shepherd KyoungSoo Park for their helpful feedback. This research was supported in part by US National Science Foundation Awards (TWC 1314857 and NeTS 1526635), the European Research Council (ERC Synergy imPACT 610150), and the German Science Foundation (DFG CRC 1223).

References

[1] Abadi, M., Budiu, M., Erlingsson, U., and Ligatti, J. Control-flow integrity. In *Proceedings of the 12th ACM Conference on Computer and Communications Security (CCS)* (2005), pp. 340–353.

[2] Aviram, A., Weng, S.-C., Hu, S., and Ford, B. Efficient system-enforced deterministic parallelism. In *Proceedings of the 9th USENIX Conference on Operating Systems Design and Implementation* (Berkeley, CA, USA, 2010), OSDI'10, USENIX Association, pp. 193–206.

[3] Banga, G., Druschel, P., and Mogul, J. C. Resource containers: A new facility for resource management in server systems. In *Proceedings of the Third Symposium on Operating Systems Design and Implementation* (Berkeley, CA, USA, 1999), OSDI '99, USENIX Association, pp. 45–58.

[4] Belay, A., Bittau, A., Mashtizadeh, A., Terei, D., Mazières, D., and Kozyrakis, C. Dune: Safe user-level access to privileged CPU features. In *Presented as part of the 10th USENIX Symposium on Operating Systems Design and Implementation (OSDI 12)* (Hollywood, CA, 2012), USENIX, pp. 335–348.

[5] Bittau, A., Marchenko, P., Handley, M., and Karp, B. Wedge: Splitting applications into reduced-privilege compartments. In *Proceedings of the 5th USENIX Symposium on Networked Systems Design and Implementation* (Berkeley, CA, USA, 2008), NSDI'08, USENIX Association, pp. 309–322.

[6] Boyd-Wickizer, S., Chen, H., Chen, R., Mao, Y., Kaashoek, F., Morris, R., Pesterev, A., Stein, L., Wu, M., Dai, Y., Zhang, Y., and Zhang, Z. Corey: An operating system for many cores. In *8th USENIX Symposium on Operating Systems Design and Implementation (OSDI)* (2008).

[7] CERT Vulnerability Note VU#720951: OpenSSL TLS heartbeat extension read overflow discloses sensitive information. `http://www.kb.cert.org/vuls/id/720951`.

[8] Chase, J. S., Levy, H. M., Feeley, M. J., and Lazowska, E. D. Sharing and protection in a single-address-space operating system. *ACM Trans. Comput. Syst. 12*, 4 (Nov. 1994), 271–307.

[9] Chen, Y., Reymondjohnson, S., Sun, Z., and Lu, L. Shreds: Fine-grained execution units with private memory. *2016 IEEE Symposium on Security and Privacy, SP 2016, San Jose, CA, USA, May 23-25, 2015* (2016), 20–37.

[10] Dieter, W. R., and Lumpp, Jr., J. E. User-level checkpointing for LinuxThreads programs. In *Proceedings of the FREENIX Track: 2001 USENIX Annual Technical Conference* (Berkeley, CA, USA, 2001), USENIX Association, pp. 81–92.

[11] Durumeric, Z., Kasten, J., Li, F., Amann, J., Beekman, J., Payer, M., Weaver, N., Halderman, J. A., Paxson, V., and Bailey, M. The matter of Heartbleed. In *ACM Internet Measurement Conference (IMC)* (2014).

[12] El Hajj, I., Merritt, A., Zellweger, G., Milojicic, D., Achermann, R., Faraboschi, P., Hwu, W.-m., Roscoe, T., and Schwan, K. SpaceJMP: programming with multiple virtual address spaces. In *Proceedings of the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems* (New York, NY, USA, 2016), ASPLOS '16, ACM, pp. 353–368.

[13] Ford, B., and Lepreau, J. Evolving Mach 3.0 to a migrating thread model. In *Proceedings of the USENIX Winter 1994 Technical Conference on USENIX Winter 1994 Technical Conference* (Berkeley, CA, USA, 1994), WTEC'94, USENIX Association.

[14] Google Caja Team. Google-Caja: A source-to-source translator for securing javascript-based web.

[15] Heiser, G., Elphinstone, K., Vochteloo, J., Russell, S., and Liedtke, J. The Mungi single-address-space operating system. *Softw. Pract. Exper. 28*, 9 (July 1998), 901–928.

[16] Intel Corp. *Intel 64 and IA-32 Architectures Software Developer's Manual: Vol. 3D*, June 2016.

[17] Kuznetsov, V., Szekeres, L., Payer, M., Candea, G., Sekar, R., and Song, D. Code-pointer integrity. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI)* (2014), pp. 147–163.

[18] Lindstrom, A., Rosenberg, J., and Dearle, A. The grand unified theory of address spaces. In *Proceedings of the Fifth Workshop on Hot Topics in Operating Systems (HotOS-V)* (Washington, DC, USA, 1995), HOTOS '95, IEEE Computer Society.

[19] Litzkow, M., Tannenbaum, T., Basney, J., and Livny, M. Checkpoint and migration of UNIX processes in the Condor distributed processing system. Tech. Rep. UW-CS-TR-1346, University of Wisconsin—Madison CS Department, April 1997.

[20] Mambretti, A., Onarlioglu, K., Mulliner, C., Robertson, W., Kirda, E., Maggi, F., and Zanero, S. Trellis: Privilege Separation for Multi-User Applications Made Easy. In *International Symposium on Research in Attacks, Intrusions and Defenses (RAID)* (Sept. 2016).

[21] McKusick, M. K., and Neville-Neil, G. V. *The Design and Implementation of the FreeBSD Operating System*. Pearson Education, 2004.

[22] Mettler, A., Wagner, D., and Close, T. Joe-e: A security-oriented subset of java. In *NDSS* (2010), vol. 10, pp. 357–374.

[23] Miller, M. *Robust composition: Towards a unified approach to access control and concurrency control*. PhD thesis, Johns Hopkins University, 2006.

[24] Palmer, G. The case for thread migration: Predictable IPC in a customizable and reliable OS. In *Proceedings of the Workshop on Operating Systems Platforms for Embedded Real-Time applications (OSPERT '10)* (2010).

[25] Patrignani, M., Agten, P., Strackx, R., Jacobs, B., Clarke, D., and Piessens, F. Secure compilation to protected module architectures. *ACM Transactions on Programming Languages and Systems 37*, 2 (Apr. 2015).

[26] Plank, J. S., Beck, M., Kingsley, G., and Li, K. Libckpt: Transparent checkpointing under Unix. In *Usenix Winter Technical Conference* (January 1995), pp. 213–223.

[27] Steinberg, U., and Kauer, B. Nova: A microhypervisor-based secure virtualization architecture. In *Proceedings of the 5th European Conference on Computer Systems* (2010), EuroSys '10, pp. 209–222.

[28] The PHP Group. FastCGI Process Manager (FPM). `http://php.net/manual/en/install.fpm.php`, 2016.

[29] Wahbe, R., Lucco, S., Anderson, T. E., and Graham, S. L. Efficient software-based fault isolation. *SIGOPS Oper. Syst. Rev. 27*, 5 (Dec. 1993), 203–216.

[30] Watson, R. N. M., Anderson, J., Laurie, B., and Kennaway, K. A taste of Capsicum: Practical capabilities for unix. *Commununications of the ACM 55*, 3 (Mar. 2012).

[31] Watson, R. N. M., Woodruff, J., Neumann, P. G., Moore, S. W., Anderson, J., Chisnall, D., Dave, N. H., Davis, B., Gudka, K., Laurie, B., Murdoch, S. J., Norton, R., Roe, M., Son, S., and Vadera, M. CHERI: A hybrid capability-system architecture for scalable software compartmentalization. In *2015 IEEE Symposium on Security and Privacy, SP 2015, San Jose, CA, USA, May 17-21, 2015* (2015), pp. 20–37.

[32] Witchel, E., Cates, J., and Asanović, K. Mondrian memory protection. In *Proceedings of the 10th International Conference on Architectural Support for Programming Languages and Operating Systems* (New York, NY, USA, 2002), ASPLOS X, ACM, pp. 304–316.

[33] Witchel, E., Rhee, J., and Asanovic, K. Mondrix: Memory isolation for Linux using Mondriaan memory protection. In *Proceedings of the 20th Symposium on Operating Systems Principles (SOSP '05)* (Brighton, UK, October 2005).

[34] Woodruff, J., Watson, R. N., Chisnall, D., Moore, S. W., Anderson, J., Davis, B., Laurie, B., Neumann, P. G., Norton, R., and Roe, M. The CHERI capability model: Revisiting RISC in an age of risk. In *Proceeding of the 41st Annual International Symposium on Computer Architecuture* (Piscataway, NJ, USA, 2014), ISCA '14, IEEE Press, pp. 457–468.

[35] Yee, B., Sehr, D., Dardyk, G., Chen, J. B., Muth, R., Ormandy, T., Okasaka, S., Narula, N., and Fullager,

N. Native Client: A sandbox for portable, untrusted x86 native code. *2009 IEEE Symposium on Security and Privacy, SP 2016, Berkeley, CA, USA, May 17-20, 2009* (2016), 79–93.

[36] ZEND. MVC Skeleton Application. `https://framework.zend.com/downloads/skeleton-app`, 2016.

[37] ZHANG, L., CHOFFNES, D., DUMITRAŞ, T., LEVIN, D., MISLOVE, A., SCHULMAN, A., AND WILSON, C. Analysis of SSL Certificate Reissues and Revocations in the Wake of Heartbleed. In *ACM Internet Measurement Conference (IMC)* (2014).

[38] ZHONG, H., AND NIEH, J. CRAK: Linux checkpoint/restart as a kernel module. Tech. Rep. CUCS-014-01, Columbia University CS Department, November 2001.

Altruistic Scheduling in Multi-Resource Clusters

Robert Grandl[1], Mosharaf Chowdhury[2], Aditya Akella[1], Ganesh Ananthanarayanan[3]
[1] *University of Wisconsin-Madison* [2]*University of Michigan* [3]*Microsoft*

Abstract

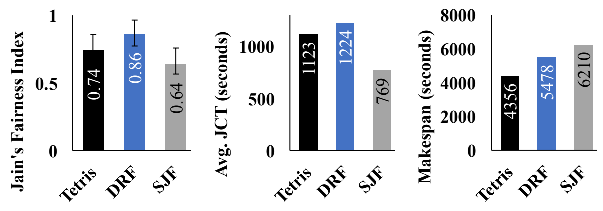
Given the well-known tradeoffs between fairness, performance, and efficiency, modern cluster schedulers often prefer instantaneous fairness as their primary objective to ensure performance isolation between users and groups. However, instantaneous, short-term convergence to fairness often does not result in noticeable long-term benefits. Instead, we propose an *altruistic*, long-term approach, CARBYNE, where jobs yield fractions of their allocated resources without impacting their own completion times. We show that leftover resources collected via altruisms of many jobs can then be rescheduled to further secondary goals such as application-level performance and cluster efficiency without impacting performance isolation. Deployments and large-scale simulations show that CARBYNE closely approximates the state-of-the-art solutions (e.g., DRF [27]) in terms of performance isolation, while providing 1.26× better efficiency and 1.59× lower average job completion time.

1 Introduction

Resource scheduling remains a key building block of modern data-intensive clusters. As data volumes increase and the need for analysis through multiple lenses explode [1, 2, 12, 23, 34, 41, 43, 45, 55], diverse coexisting jobs from many users and applications contend for the same pool of resources in shared clusters.

Consequently, today's cluster schedulers [9,17,33,51] have to deal with multiple resources [14, 20, 27, 29, 38], consider jobs with complex directed acyclic graph (DAG) structures [19,29,55], and allow job-specific constraints [15,28,35,54,56]. Schedulers must provide performance isolation between different users and groups through fair resource sharing [9, 16, 17, 27, 29, 36, 56], while ensuring performance (low job completion times) and efficiency (high cluster resource utilization).

However, simultaneously optimizing fairness, performance, and efficiency is difficult. Figure 1 demonstrates the tradeoff space by comparing three schedulers – dominant resource fairness (DRF) [27] for multi-resource fairness, shortest-job-first (SJF) [26] for minimizing the average job completion time (JCT), and Tetris [29] for increasing resource utilization – implemented in Apache YARN [51]: each scheduler outperforms its counterparts



(a) Inter-job fairness **(b)** Job performance **(c)** Cluster efficiency

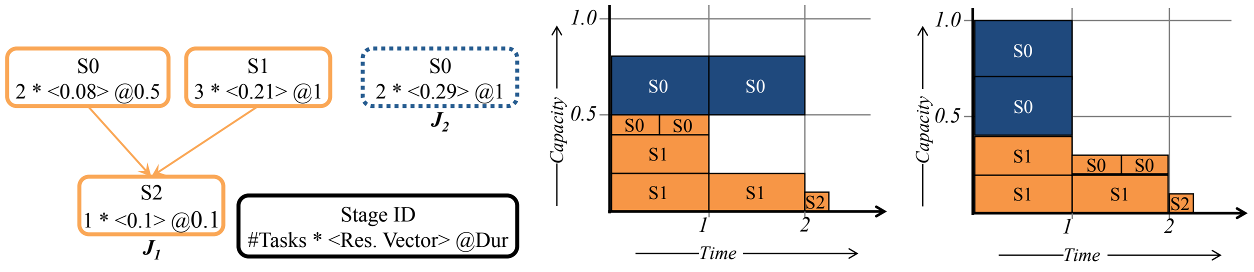
Figure 1: DRF [27], SJF [26], and Tetris [29] on a TPC-DS [7] workload on a 100-machine cluster. All jobs arrived together. Higher is better in (a),[1] while the opposite is true in (b) and (c).

only in a preferred metric and *significantly* underperforms in the secondary metrics. In practice, many production schedulers [9,10,17,33,48,51,52] settle for performance isolation as their primary objective and focus on quick convergence to *instantaneous* fair allocations, while considering performance and efficiency as *best-effort*, secondary goals, often without an explicit focus on them.

In this paper, we revisit the three-way tradeoff space: *is it possible to ensure performance isolation (i.e., fairness) and still be competitive with the best approaches for the secondary metrics (job completion time and cluster utilization)?* We highlight two key characteristics. First, distributed data-parallel jobs share an *all-or-nothing* characteristic [14, 15, 21, 56]: a parallel job cannot complete until all its tasks have completed. Because of this, aggressively using all of the fair share often does not translate into noticeable benefits in terms of secondary metrics. In fact, it can hurt the average JCT and cluster efficiency. Second, users only observe the outcome of performance isolation when their jobs complete, and they care less for instantaneous fair-share guarantees.

Thus, we propose an *altruistic*, *long-term* approach based on a simple insight: jobs yield fractions of their currently allocated resources – referred to as *leftover* resources – to be redistributed to further secondary objectives. As long as jobs can contribute resources and still complete without taking additional time, the effects of such redistribution remain transparent to users. (We

[1]In Figure 1a, we calculated the average of Jain's fairness index [37] over 60-second intervals; error bars represent the minimum and the maximum values.



(a) Two DAGs w/ their dependencies and requirements **(b)** Aggressive scheduling **(c)** Altruistic scheduling

Figure 2: Opportunities in altruistic scheduling for two DAGs contending on a single bottleneck resource. Job J_1 (orange/light/solid line) has three stages and Job J_2 (blue/dark/dotted line) has one stage (a), where the number of tasks in each stage along with their expected resource requirements and durations are shown as specified in the legend. Assuming each task can start and complete its execution in the specified duration given its required resources, (b)–(c) depict the allocations of the bottleneck resource (vertical axis) for different approaches: The average completion time for (b) traditional schedulers (i.e., single- or multi-resource fairness across jobs and efficient packing within each job) is 2.05 time units; and (c) CARBYNE with altruistic scheduling is 1.55 time units.

prove that an altruistic scheduler can guarantee this in the offline setting).

Indeed, jobs have ample opportunities for altruisms in production (§2). For example, 50% of the time, at least 20% of the allocated resources can be used as leftover (i.e., yielded altruistically) at Microsoft's production analytics clusters. Given a fixed amount of resources, a job may be altruistic whenever it cannot simultaneously schedule all of its runnable tasks – i.e., when it can choose which tasks to schedule at that instant and which ones to schedule later. As clusters are shared between more users and DAG complexity (e.g., the number of barriers, total resource requirements, and critical path length [39]) increases, the amount of leftover resources increases too.

Leftover resources collected via altruisms of many jobs introduce an *additional degree of freedom* in cluster scheduling. We leverage this flexibility in CARBYNE that decides how to determine and redistribute the leftover, so as to improve one or both of performance and efficiency metrics as the cluster operator sees fit, while providing the same level of user-observable performance isolation as the state-of-the-art (§3). Specifically, given a share of the cluster resources (computed by a fair-sharing scheme like DRF), CARBYNE's intra-job scheduler computes task schedules further into the future (since it knows the DAG of the job upfront). This allows each job to compute, at any given time, which runnable tasks to schedule and how much of its fair share to contribute altruistically. CARBYNE's inter-job scheduler uses the resulting leftover resources to: (a) preferentially schedule tasks from jobs that are closest to completion; then (b) pack remaining tasks to maximize efficiency. The order of (a) and (b) can be reversed to prioritize efficiency over JCTs. Note that our focus is not on improving the scheduling heuristics for the individual metrics themselves, but rather on combining them with altruism. To that end, our solution can work with any of the scheduling heuristics designed for fairness (say, capacity scheduler [5]), completion time, and cluster efficiency (say, DAGPS [30, 31]).

Prior work have also attempted to simultaneously meet multiple objectives [18, 21, 26, 29–31, 38, 47]. The main difference is that given some resources (e.g., according to some fair allocation) prior approaches (notably [29–31]) adopt eager scheduling of tasks, whereas CARBYNE schedules altruistically by delaying tasks in time as much as possible to accommodate those that are in greater need of running now. As our examples in Section 2 and results in Section 5 show, such an altruism-based approach offers better control over meeting global objectives. In particular, compared to [29–31], CARBYNE is better at meeting fairness guarantees, while offering similar performance along other objectives.

We have implemented CARBYNE as a pluggable scheduler on Apache YARN [51] (§4). Any framework that runs on YARN can take advantage of CARBYNE. We deployed and evaluated CARBYNE on 100 bare-metal machines using TPC-DS [7], TPC-H [8], and BigBench [4] queries and also by replaying production traces from Microsoft and Facebook in trace-driven simulations (§5). In deployments, CARBYNE provides 1.26× better efficiency and 1.59× lower average completion time than DRF, while closely approximating it in fairness. In fact, CARBYNE's performance and efficiency characteristics are close to that of SJF and Tetris, respectively. Only 4% jobs suffer more than 0.8× slowdown (with the maximum slowdown being 0.62×) as CARBYNE temporally reorganizes tasks to improve all three metrics. We observed slightly higher benefits in simulations for complex DAGs as well as for simple MapReduce jobs. Furthermore, we found that CARBYNE performs well even in the presence of misestimations of task demands (needed for packing) and even when some jobs are not altruistic.

	# Stages		# Barriers		Input Work	
Workload	50th	95th	50th	95th	50th	95th
Microsoft	13	121	4	13	34%	85%
TPC-DS	8	23	1	4	11%	88%
TPC-H	8	12	2	4	46%	82%
BigBench	7	19	2	6	24%	70%

Table 1: Structural diversity in various workloads. Each group of columns reads out percentile distributions.

	CP Length		# Disjoint Paths	
Workload	50th	95th	50th	95th
Microsoft	7	17	6	34
TPC-DS	5	8	4	15
TPC-H	5	7	3	7
BigBench	5	8	2	10

Table 2: Diversity in length of the critical path (CP) and the number of disjoint paths in DAGs across various workloads.

2 Motivation

This section demonstrates benefits of altruistic scheduling using an example (§2.1) and quantitatively analyzes altruism opportunities (§2.2) in DAG workloads.

2.1 Illustration of Altruism

Modern schedulers focus on *instantaneous* (i.e., short-term) optimizations and take greedy decisions. We hypothesize that the focus on the short-term restricts their flexibility in optimizing secondary objectives. Instead, if we relax them via *short-term altruisms* and focus on long-term optimizations, we can enable disproportionately larger flexibility during scheduling, which, in turn, can translate into significant improvements.

Consider Figure 2 that compares two schedules assuming both jobs arrive at the same time. Existing schedulers (Figure 2b) perform independent scheduling both across and within jobs, resulting in an average completion time of 2.05 time units (by allocating equal share of resources to the jobs). This holds for *any* combination of today's schedulers – max-min [36] or dominant resource fairness (DRF) [27] across different jobs and breadth-first order [2, 55], critical path method (CPM) [39, 40], or packing [29] within each job – because they are independent by design to optimize for short-term objectives.

In contrast, CARBYNE takes a long-term approach (Figure 2c), where the intra-job scheduler of J_1 altruistically gives up some of its resources (since S2 in J_1 is a barrier), which can better serve J_2. As long as tasks in stages $S0$ and $S1$ finish by 2 time units, J_1's completion time will not change. By ignoring short-term greed, J_1 can improve J_2's completion time, resulting in an average completion time of 1.55 time units (1.3× better). [2]

Note that this example considers only one resource, only two jobs, and simplistic DAGs. As complexity increases across all these dimensions, CARBYNE can be even more effective (§5).

2.2 Opportunities in Analytics DAGs

In this section, we analyze the potential for altruism. We analyzed query DAGs from a large Microsoft production trace, TPC-DS [7], TPC-H [8], and BigBench [4] benchmarks. We first quantify the amount of leftover resources for altruism and then correlate it with relevant DAG properties. For a detailed analysis of all DAG properties in production, we refer interested readers to [30, 31].

How much resources can be considered as leftover and used for altruistic scheduling? To answer this, we computed the fraction of allocated compute, memory, disk in/out, and network in/out bandwidths that could have been reallocated without affecting any job's completion time at Microsoft for each scheduling event such as job and task arrivals and completions. We found that across thousands of scheduling events, 50% of the time, at least 20% of the allocated resources – $\langle 35\%, 43\%, 24\%, 21\%, 28\%, 24\% \rangle$ for the resources listed above – could be used as leftover and rescheduled via altruistic scheduling.

We next analyze the properties of the jobs that matter the most for altruism: total stages, number of stages with multiple parents (i.e., barriers), length of the DAG's critical path, and the number of disjoint paths in the DAG.

Stage-Level Observations The number of stages in a DAG provides an approximation of its complexity. In our traces, a DAG has at least 7 stages at median and up to 23 at the 95th percentile (Table 1). These numbers are significantly higher for production traces.

Recall that CARBYNE is altruistic without hampering JCT. A key factor dictating this is the number of barriers, i.e., stages whose children must wait until they finish. Examples include aggregation and join stages, or even final output stages. We observed the presence of multiple barriers in most DAGs (Table 1); also quantified in [15].

Path-Level Observations Next, we considered the length of the critical path of the DAG [39] as well as the number of disjoint paths. We defined each sequence of stages that can run in parallel with other sequences of different stages as a disjoint path. Table 2 shows that the median length of the critical path of DAGs is 5 in these workloads, and many DAGs have multiple disjoint paths, further enabling altruistic scheduling; [30, 31] has a detailed description and quantification of DAG critical paths.

[2]For this example, packing across job boundaries results in an average JCT of 1.55 time units. However, as shown in Figure 1b, packing does not always lead to the best average JCT, and it is not difficult to construct a similar example.

Correlating DAG properties and leftover resources To understand which of these DAG properties have the highest impact, we calculated correlations over time between the amount of leftover resources via CARBYNE and variabilities in each of the aforementioned attributes. We found that the number of barriers in a DAG has the highest positive correlation (0.75). The more barriers a DAG contains, the more opportunities it has to contribute. For example, in Figure 2c, the barrier $S2$ of DAG J_1 enabled its altruism. The number of stages, critical path length, and the number of disjoint paths also have high positive correlations of 0.66, 0.71 and 0.57, respectively.

3 Altruistic Multi-Resource Scheduling

In this section, we present an online altruistic multi-resource DAG scheduler. First, we define the problem along with our assumptions (§3.1). Next, we discuss desirable properties of an ideal DAG scheduler and associated tradeoffs (§3.2). Based on our understanding, we develop an offline altruistic scheduler in two steps (§3.3): determining how much a job can contribute to leftover and deciding how to distribute the leftover or yielded resources to other jobs. Finally, we analyze why the offline solution works well in the online scenario as well (§3.4).

3.1 Problem Statement

Given a collection of jobs – along with information about individual tasks' expected multi-resource requirements, durations, and DAG dependencies – we must schedule them such that each job receives a fair share of cluster resources, jobs complete as fast as possible, and the schedule is work-conserving. All information about individual jobs are unknown prior to their arrival.

3.2 Complexity and Desirable Properties

Offline DAG scheduling is NP-complete [25] for all the objectives[3] – fairness, performance, and efficiency – even when the entire DAG and completion times of each of its stages are known. In fact, polynomial-time optimal DAG scheduling algorithms exist for only three simple cases [22, 42], none of which are applicable to DAG scheduling in multi-resource clusters.

Because achieving optimality in all of the aforementioned objectives is impossible due to their tradeoffs [18, 21, 26, 29, 38, 47], we want to build a scheduler that improves performance and efficiency without sacrificing performance isolation. In the online case, we expect an ideal such scheduler to satisfy the following goals:

- *Fast completion:* Each DAG should complete as fast as possible.
- *Work conservation:* Available resources should not remain unused.
- *Starvation freedom:* No DAG should starve for arbitrarily long periods.[4]

3.3 Offline Altruistic Scheduling

Cluster schedulers typically operate in two levels: inter- and intra-job; i.e., between jobs and between tasks of each job. However, we want to consider three distinct scheduling components – the two above, and leftover scheduling. To this end, we first identify an *intermediate* degree of freedom in scheduling, and we discuss how to leverage that.

3.3.1 Solution Approach

Consider the offline problem of scheduling $|\mathbb{J}|$ DAG jobs ($\mathbb{J} = \{J_1, J_2, \ldots, J_{|\mathbb{J}|}\}$) that arrived at time 0. We start by observing that given a fixed share ($\overrightarrow{A_k} = \langle a_k^1, a_k^2, \ldots a_k^{|\overrightarrow{R}|} \rangle$) of $|\overrightarrow{R}|$ resources ($\overrightarrow{R} = \langle R^1, R^2, \ldots, R^{|\overrightarrow{R}|} \rangle$) by an inter-job scheduler, a DAG will take a minimum of amount of time (T_k for job J_k) to complete all its tasks, given their dependencies. However, as long as its allocation does not decrease – true in the offline case – it can decide *not to be aggressive* in using resources and still complete by T_k. Formally, we prove the following:

Theorem 3.1 *Altruism will not inflate any job's completion time in the offline case – i.e., unless new jobs arrive or existing jobs depart – for any inter-job scheduler.*

The proof follows from the fact that none of the T_k's invariants – namely, resource requirements, DAG structure, and allocation of J_k – change in the offline case.

We refer to resources that are not immediately required as *leftover* resources ($\overrightarrow{L_k} = \langle l_k^1, l_k^2, \ldots l_k^{|\overrightarrow{R}|} \rangle$), and we consider contributing to $\overrightarrow{L_k}$ to be an *altruistic* action. For example, in Figure 2c, J_1 can potentially have 0.29 units of leftover resources at time 0. Assuming its fixed resource share of 0.5 units, it would still be able to complete in 2.1 time units by delaying one more task from stage $S1$ to start at time 1. Note that J_1 is instead running that task at time 0 to ensure work conservation.

Given that multiple jobs can contribute leftover resources (§2.2), if we combine ($\overrightarrow{\mathbb{L}} = \sum_k \overrightarrow{L_k}$) and redistribute them across all running jobs, we can create new opportunities for improving the secondary metrics. We can now reformulate cluster scheduling as the following three questions:

1. *how to perform inter-job scheduling to maximize the amount of leftover resources?*
2. *how should an intra-job scheduler determine how much a job should contribute to leftover?* and
3. *how to redistribute the leftover across jobs?*

[3]Whether multi-resource fair DAG scheduling is NP-complete is unknown. DRF [27] results were proven for jobs where *all* tasks have the same resource requirements, which is not the case in multi-stage DAGs.

[4]We do not aim for stronger goals such as *guaranteeing bounded starvation*, because providing guarantees require resource reservation along with admission control. Even DRF does not provide any guarantees in the online case.

By addressing each one at each scheduling granularity – inter-job, intra-job, and leftover – we design an altruistic DAG scheduler (Pseudocode 1) that can compete with the best and outperform the rest in all three metrics (§5).

CARBYNE invokes Pseudocode 1 on job arrival, job completion, as well as task completion events. For each event, it goes through three scheduling steps to determine the tasks that must be scheduled at that point in time. While we use DRF [27], SRTF, and Tetris [29] for concrete exposition, CARBYNE can also equivalently use other schedulers such as DAGPS [30, 31] (instead of Tetris) for packing and capacity scheduler [5] (instead of DRF) for fairness.

3.3.2 Increasing Leftover via Inter-Job Scheduling

Modern clusters are shared between many groups and users [9, 10, 16, 17, 33, 48, 51, 52]. Consequently, the primary goal in most production clusters is performance isolation through slot-based [9, 10] or multi-resource [16, 27] fairness. We use a closed-form version of DRF [46] for inter-job scheduling in CARBYNE. Interestingly, because a single- or multi-resource fair scheduler enforces resource sharing between *all* running jobs, it elongates individual job completion times the most (compared to schedulers focused on other primary objectives). Consequently, fair schedulers provide the most opportunities for altruistic scheduling.

3.3.3 Determining Leftover for Individual Jobs

Given some resource share $\overrightarrow{A_k}$, CARBYNE's intra-job scheduler aims to maximize the amount of leftover resources from each job. We schedule only those tasks that *must* start running for J_k to complete within the next T_k duration, and we altruistically donate the rest of the resources for redistribution. Computing this is simple: we simply perform a reverse/backward packing of tasks from T_k to current time, packing tasks in time as close to T_k as possible, potentially leaving more resources available at earlier times (lines 17–21 in Pseudocode 1). For example, in Figure 2, stage $S2$ of job J_1 can only start after 2 time units. Hence, CARBYNE can postpone both tasks from $S0$ and one task from $S1$ until at least the first time unit, donating 0.29 units to leftover resources (0.08 for $S0$'s tasks and 0.21 for $S1$'s task). Similarly, job J_2 donates 0.21 units to leftover resources (its fair share of 0.5 less the resource used by one task in its $S0$ of 0.29), making it a sum of 0.5 leftover resource units.

Reverse packing uses the same principle as the intra-coflow scheduler used in Varys [21], where most flows in a coflow are slowed down so that all of them finish together with the longest running one. However, CARBYNE considers multiple resource types as well as dependencies, and unlike Varys, it does not hog CPU and memory.

A sophisticated packer (like DAGPS [30, 31]) can better increase leftover resources. As opposed to our reverse packer that only considers the parents of currently runnable tasks, it could consider the entire DAG and its dependencies. It could also account for the nature of dependencies between stages (e.g., many-to-one, all-to-all). The more CARBYNE can postpone tasks into the future without affecting a DAG's completion time, the more leftover resources it has for altruistic scheduling.

Pseudocode 1 Altruistic DAG Scheduling

```
1: procedure SCHEDULE(Jobs $\mathbb{J}$, Resources $\overrightarrow{R}$)
2:     $\{\overrightarrow{A_k}\}$, $\overrightarrow{\mathbb{L}}$ = INTERJOBSCHEDULER($\mathbb{J}$, $\overrightarrow{R}$)
3:     for all $J_k \in \mathbb{J}$ do
4:         $\overrightarrow{\mathbb{L}}$ += INTRAJOBSCHEDULER($J_k$, $\overrightarrow{A_k}$)
5:     end for
6:     LEFTOVERSCHEDULER($\mathbb{J}$, $\overrightarrow{\mathbb{L}}$)
7: end procedure

8: procedure INTERJOBSCHEDULER(Jobs $\mathbb{J}$, Resources $\overrightarrow{R}$)
9:     $\overrightarrow{L_{\text{DRF}}} = \overrightarrow{R}$     ▷ $\overrightarrow{L_{\text{DRF}}}$ tracks total unalloc. resources
10:    Calculate $\overrightarrow{A_k}$ using closed-form DRF [46] for all $J_k$
11:    for all $J_k \in \mathbb{J}$ do
12:        $\overrightarrow{L_{\text{DRF}}}$ -= $\overrightarrow{A_k}$
13:    end for
14:    return $\{A_k\}$ and $\overrightarrow{L_{\text{DRF}}}$
15: end procedure

16: procedure INTRAJOBSCHEDULER(Job $J_k$, Alloc. $\overrightarrow{A_k}$)
17:    Given $\overrightarrow{A_k}$, calculate $T_k$ using Tetris [29]
18:    Reverse parent-child task dependencies in $J_k$
19:    Calc. task start times Rev($t_k^j$) in the reversed DAG
20:    Actual($t_k^j$) = Rev($t_k^j$) + $T_k$ − Dur($t_k^j$)
21:    $\mathbb{T}_k = \{t_k^j : \text{Actual}(t_k^j) == 0\}$ ▷ Tasks that must start
22:    Schedule $t_k^j$; $\forall t_k^j \in \mathbb{T}_k$
23:    $\overrightarrow{L_k} = \overrightarrow{A_k} - \sum \text{Req}(t_k^j)$; $t_k^j \in \mathbb{T}_k$
24:    return $\overrightarrow{L_k}$
25: end procedure

26: procedure LEFTOVERSCHEDULER(Jobs $\mathbb{J}$, Leftover $\overrightarrow{\mathbb{L}}$)
27:    $\mathbb{J}'$ = SORT_ASC ($\mathbb{J}$) in the SRTF order
28:    for all $J_k \in \mathbb{J}'$ do
29:        Schedule runnable tasks to maximize $\overrightarrow{\mathbb{L}}$ usage
30:    end for
31: end procedure
```

3.3.4 Redistribution via Leftover Scheduling

Given the pool of leftover resources $\overrightarrow{\mathbb{L}}$ from all jobs (0.5 units in the running example), leftover scheduling has two goals:

- *Minimizing the average JCT* by scheduling tasks from jobs that are closest to completion using Shortest-Remaining-Time-First (SRTF[5]). This schedules the task from stage $S0$ of J_2 in the first time step, leaving 0.21 units of total leftover.

[5]This is the shortest amount of work first, as in Tetris [29].

- *Maximizing efficiency* by packing as many unscheduled tasks as possible – i.e., using Tetris [29]. This results in one more task of stage $S1$ of J_1 to be scheduled, completing leftover redistribution.

3.3.5 Properties of the Offline Scheduler

The former action of the leftover scheduler *enables fast job completions*, while the latter *improves work conservation*. At the same time, because the intra-job scheduler ensures that T_k values are not inflated, the overall scheduling solution maintains the fairness characteristics of DRF and *provides the same performance isolation* and starvation freedom. In summary, the offline CARBYNE scheduler satisfies all the desirable properties.

3.4 From Offline to Online

In the online case, new jobs can arrive before T_k and decrease J_k's resource share, breaking one of the invariants of **Theorem 3.1**. For example, given N jobs and one resource, J_k receives $\frac{1}{N}$-th of the resource; if M more jobs arrive, its share will drop to $\frac{1}{N+M}$-th. If J_k was altruistic earlier, as new jobs arrive, it cannot complete within the previously calculated T_k time units any more.

3.4.1 Analysis of Online Altruistic Scheduling

In practice, even in the online case, we observe marginal impacts on only a handful of jobs (§5.2.3). This is because (i) production clusters run hundreds of jobs in parallel; and (ii) resource requirements of an individual task is significantly smaller than the total capacity of a cluster.

Consider a single resource with total capacity R, and assume that tasks across *all* jobs take the same amount of time (t) and resource ($r \ll R$). Given N jobs, job J_k received $A_k = \frac{1}{N}$-th share of the single resource, and it is expected to complete in T_k time units while scheduling *at most* $\frac{A_k}{r}$ tasks per time unit.

Now consider M new jobs arrive when J_k is $T'_k (< T_K)$ time units away from completion. Their arrival decreases J_k's share to $A'_k = \frac{1}{N+M}$-th, when J_k must be able to schedule tasks at $\frac{A_k}{r}$ rate to complete within T_k.

Assuming all resources being used by the running jobs, the rate at which J_k can schedule tasks is the rate at which tasks finish, i.e., $\frac{t}{R/r}$. Meaning, J_k is expected to take $\frac{A'_k t}{R/r}$ time units to schedule all tasks.

The additional time to schedule the remaining tasks will be negligible compared to remaining time T'_k as long as tasks finish uniformly randomly over time[6] and

$$T'_k \gg \frac{1}{N+M}\frac{r}{R}t$$

The above holds for most production clusters because typically $N \gg 1$ and $R \gg r$.

[6]This holds true in large production clusters [54].

3.4.2 Bounding Altruism

As a failsafe to prevent jobs from being repetitively punished for altruism, CARBYNE provides a uniformly distributed probability $P(Altruism)$. It dictates with what probability a job will yield its resources during any scheduling event. It is set to 1 by default – i.e., jobs yield resources whenever possible – causing less than 4% of the jobs to suffer at most 20% slowdown (§5.2.3). Decreasing it results in even fewer jobs to suffer slowdown at the expense of lower overall improvements (§5.4.3). In an adversarial environment, one can avoid altruism altogether by setting $P(Altruism) = 0$, which reduces CARBYNE to DRF.

3.5 Discussion

Modern clusters must consider constraints such as data locality and address runtime issues such as stragglers and task failures. CARBYNE is likely to have minimal impact on these aspects.

Data Locality Because disk and memory locality significantly constrain scheduling choices [14, 23, 28], altruistically giving up resources for a data-local task may adversely affect it in the future. However, delay scheduling [54] informs us that waiting even a small amount of time can significantly increase data locality. An altruistically delayed data-local task is likely to find data locality when it is eventually scheduled.

Straggler Mitigation Techniques such as speculative execution are typically employed toward the end of a job to mitigate outliers [13, 15, 23, 56]. CARBYNE is likely to prioritize speculative tasks during leftover scheduling because it selects jobs in the SRTF order.

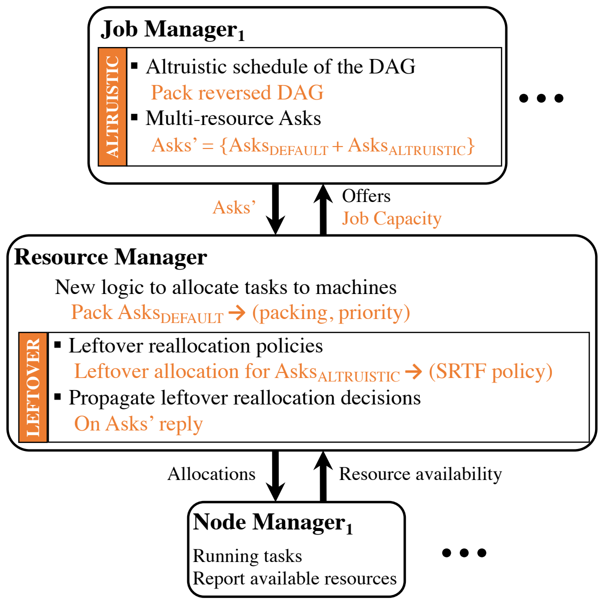
Handling Task Failures Similar to existing solutions, CARBYNE does not distinguish between new and restarted tasks. However, in case of task failures, CARBYNE must recalculate the completion estimation (T_k) of the corresponding job J_k.

4 Design Details

In this section, we describe how to enable altruism in different cluster schedulers, explain how we have implemented CARBYNE in YARN and Tez, and discuss how we estimate task resource requirements.

4.1 Enabling Altruism in Clusters Schedulers

Enabling altruistic scheduling requires two key components. First, a local *altruistic resource management* module in each application must determine how much resources it can yield. Second, the global cluster scheduler must implement a *leftover resource management* module to reallocate the yielded resources from all applications. RPC mechanisms between the two schedulers may need to be expanded to propagate the new information.



Job Manager$_1$
ALTRUISTIC
▪ Altruistic schedule of the DAG
Pack reversed DAG
▪ Multi-resource Asks
Asks' = {Asks$_{DEFAULT}$ + Asks$_{ALTRUISTIC}$}
Asks'
Offers
Job Capacity
Resource Manager
New logic to allocate tasks to machines
Pack Asks$_{DEFAULT}$ → (packing, priority)
LEFTOVER
▪ Leftover reallocation policies
Leftover allocation for Asks$_{ALTRUISTIC}$ → (SRTF policy)
▪ Propagate leftover reallocation decisions
On Asks' reply
Allocations
Resource availability
Node Manager$_1$
Running tasks
Report available resources

Figure 3: Multi-resource scheduling in a data-parallel cluster. CARBYNE-related changes are shown in orange.

In case of monolithic schedulers such as YARN,[7] individual job managers (e.g., Spark/Tez master) implement the altruism module. The leftover management module is implemented in the cluster-wide resource manager. For two-level schedulers such as Mesos [33], Mesos master's global allocation module should manage the leftover resource management, while the altruistic module can be implemented at the framework-level such as Spark or MPI. Similar architectural principles apply for shared-state schedulers (e.g., Omega [48]) too.

4.2 CARBYNE System

In the following, we describe how we have implemented the two modules to provide altruism. Figure 3 depicts the core pieces of typical parallel cluster schedulers today as well as the core new functionality we add to integrate CARBYNE, marked in orange. We modified and added 1400 lines of code to implement CARBYNE on YARN.

Primer on Data-Parallel Cluster Scheduling Typically, today's data-parallel cluster schedulers divide the scheduling procedure into three parts.

A *node manager* (NM) runs on every machine in the cluster, and it is responsible for running tasks and reporting available resources.

For each job, a *job manager* or Application Master (AM) runs on some machine in the cluster and holds job context information regarding various types of tasks to be executed (pending/in-progress/completed), their dependencies (e.g., DAG) and resource requirements (e.g., memory, CPU).

[7]Please refer to [48] on why YARN is a monolithic, not two-level, scheduler.

A cluster-wide *resource manager* (RM) receives `Ask` requests from various AMs for their pending tasks to be scheduled and information about the available resources on different machines from NMs through heartbeats. Based on this information and fairness considerations, it assigns tasks to machines. Typically, an `Ask` contains information such as preferences for data locality, the amount of resources required and priorities at which tasks should be executed. Priority is an useful mechanism to enable AMs to encode their preferences to execute one task over the other (e.g., due to ordering in the DAG or failures).

CARBYNE Implementation We have built CARBYNE as an extension to the YARN [51] and Tez [2] frameworks. To implement Pseudocode 1, we made the following modifications:

1. RPC Mechanism We enhanced the RPC protocol between YARN RM and Tez AM to propagate the total resource allocation of a job as computed at RM, according to the fairness policy enforced in the cluster and the current runnable jobs. Also, we extended the `Ask` data structure to support `Asks` of different types (`Asks`$_{\text{DEFAULT}}$ for tasks that it *must* run in order to not be slow down due to altruism and `Asks`$_{\text{ALTRUISTIC}}$ for tasks that it *may* run if the job scheduler tries to use all the allocated resources) as well as other relevant information (e.g., task's demand estimates across multiple dimensions, task's priority, remaining work etc.).

2. Tez Job Manager Given the most recent resource allocation for a job received from the RM, CARBYNE-enabled Tez AM altruistically schedules the remaining DAG (i.e., unscheduled tasks and their dependencies). It implements, the `IntraJobScheduler` procedure from Pseudocode 1 to encode resource requests (`Asks`$_{\text{DEFAULT}}$) for tasks that should be scheduled via altruism. To do that, CARBYNE does reverse packing using [29] to identify the minimum set of tasks that should run as of now while respecting their dependencies, in order to not slow down the expected job completion time. In addition, it computes `Asks`$_{\text{ALTRUISTIC}}$ using the default intra-job scheduler in Tez that decides on a *breadth-first ordering* for scheduling tasks. The *optional* set of tasks are the ones that can be scheduled according to a greedy scheduler and provides to the RM additional information in order to reallocate leftover resources. While we use [29] to do reverse packing and *breadth-first ordering* to compute `Asks`$_{\text{ALTRUISTIC}}$, any other intra-job scheduling technique (e.g., DAGPS [30, 31], Critical Path Method) can be used as well. AM also includes priorities with each task that serve as "hints" to the RM as discussed below.

3. YARN's Resource Manager The scheduling process in YARN RM is triggered whenever an NM reports available resources. We updated YARN RM's matching logic to project tasks' resource requests onto available capacity. First, among the runnable jobs, it computes periodically their DRF allocation and propagates the corresponding resource allocation on the next heartbeat response to every AM. (`InterJobScheduler` procedure from Pseudocode 1). Second, it schedules tasks requests from jobs $\texttt{Asks}_{\text{DEFAULT}}$ using similar heuristics as in Tetris [29] to do packing and reduce job completion time. In addition, we encode the priorities of the tasks into these heuristics in a way similar to DAGPS [30, 31] to account for tasks dependencies in the same DAG's job as instructed by the Tez Job Manager. Packing resource requests from $\texttt{Asks}_{\text{DEFAULT}}$ according to the job AM-hinted task priorities enables the RM scheduler to enforce the altruistic schedule computed by each job AM while improving the cluster utilization. If no more $\texttt{Asks}_{\text{DEFAULT}}$ can be satisfied, the RM scheduler satisfies $\texttt{Asks}_{\text{ALTRUISTIC}}$ resource requests from jobs sorted ascending based on the amount of remaining work (emulating SRTF), until no more requests can be satisfied during this scheduling cycle (`LeftOverScheduler` procedure from Pseudocode 1). This approach prefers efficiency as a secondary objective over JCTs; however, their order can be reversed. For example, the RM scheduler can pack resource requests from $\texttt{Asks}_{\text{ALTRUISTIC}}$ and satisfies $\texttt{Asks}_{\text{DEFAULT}}$ based on the amount of remaining work.

4.3 Demand Estimation

CARBYNE relies on estimates of tasks' resource demands – across CPU, memory, disk, and the network – and their durations to make scheduling decisions. Since modern datacenters have zero or small over-subscription factors [3, 49], CARBYNE considers only access link bandwidths between a machine and its ToR switch.[8]

To estimate tasks' demands and durations, CARBYNE leverages well-established techniques such as using history of prior runs for recurring jobs [11, 24, 29] and assuming tasks from the same stage to have similar resource requirements [14, 27, 44]. We note that requiring manual annotation is also possible, and there are some promising efforts to infer task requirements from program analysis [32]; CARBYNE currently does not use such techniques. While CARBYNE performs the best with accurate estimations, we have found that using the aforementioned techniques work well in practice (§5.4.2).

5 Evaluation

We evaluated CARBYNE on a 100-machine cluster [6] using publicly available benchmarks – TPC-DS, TPC-H, and BigBench – as well as Hive traces collected from large production clusters at Microsoft and Facebook. To understand performance at a larger scale, we used a trace-driven simulator to replay task logs from the same traces. Our key findings are:

- CARBYNE can closely approximate DRF, Tetris, and SJF in terms of fairness, efficiency, and performance, respectively, in both offline and online scenarios (§5.2). Moreover, it provides $1.26\times$ better efficiency and $1.59\times$ lower average completion time than DRF.
- CARBYNE provides similar benefits in large-scale simulations, even for simple MapReduce jobs (§5.3).
- Sensitivity analysis show that CARBYNE performs even better with resource contention, and it is robust to misestimations in resource requirements and when jobs are not always altruistic (§5.4).

In the following, unless otherwise explicitly mentioned, we refer to CARBYNE as our implementation atop YARN and TEZ as described in Section 4 using Tetris as an intra-job scheduler (other than Section 5.5) and DRF for fairness.

5.1 Experimental Setup

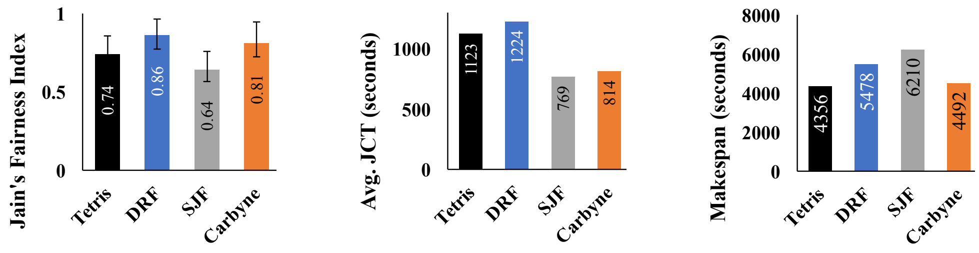
Workloads Our workloads consist of mix of jobs from public benchmarks – TPC-DS [7], TPC-H [8], BigBench [4], and from Microsoft and Facebook production traces collected from clusters with thousands of machines. Our experimental methodology is adapted from and similar to prior work [30, 31]. In each experiment run, the jobs are randomly chosen from one of the corresponding benchmark and follows a Poisson arrival distribution with average inter-arrival time of 20 seconds. Each job lasts from few minutes to tens of minutes, and we generate corresponding input sizes from GBs to hundreds of GBs. Unless otherwise noted, each experiment has 250 jobs. Microsoft workload has 30000 job DAGs with millions of tasks (§2.2). Facebook workload has 7000 jobs and $650,000$ tasks spanning six hours, and jobs use the actual arrival times from the trace. Each experiment is run three times, and we present the median.

Cluster Our experiments use 100 bare-metal servers. Each machine has 20 cores, 128 GB of memory, 128 GB SSD, a 10 Gbps NIC and runs CentOS 7. Meaning, the cluster can run up to 2000 tasks in parallel. The network has a congestion-free core.

Simulator To run CARBYNE at a larger scale and to gain more insights into CARBYNE's performance, we built a simulator that replays job traces. The simulator mimics various aspects of the logs, handling jobs with different arrival times and dependencies as well as multiple fairness and scheduling schemes.

Compared Scheduler Baselines We compare CARBYNE primarily against the following approaches:

[8]For oversubscribed networks with well-defined bottlenecks (e.g., host-toToR links), one may consider bandwidth demands on the oversubscribed links instead.



(a) Inter-job fairness **(b)** Job performance **(c)** Cluster efficiency

Figure 4: [Cluster] CARBYNE's performance against the best schemes in fairness (DRF), improvement in average JCT (SJF), and achieving cluster efficiency (Tetris) in the *offline* case as shown in Figure 1. CARBYNE approaches the best in each category and outperforms the rest. Higher is better in (a), while the opposite is true in (b) and (c).

1. **DRF**: YARN's implementation of the DRF algorithm [27] for inter-job scheduling along with the default intra-job scheduler in Tez that decides on a *breadth-first ordering* for scheduling tasks;
2. **Tetris**: Uses breadth-first-ordering for intra-job scheduling and Tetris [29] (with its fairness knob, $f \rightarrow 0$) for inter-job scheduling;
3. **SJF**: A shortest-job-first scheduler that uses Critical Path Method (CPM) to determine job duration and schedules job in the shortest-first order. We use SJF primarily as an upper-bound of CARBYNE's improvement in average job completion time.

Metrics Our primary metric to quantify performance is the improvement in the *average JCT*, computed as:

$$\text{Factor of Improvement} = \frac{\text{Duration of an Approach}}{\text{Duration of CARBYNE}} \tag{1}$$

Factor of improvement greater than 1 means CARBYNE is performing better, and vice versa.

Additionally, we use *makespan*, i.e., when all jobs in a workload completed, to measure efficiency.

Finally, to quantify fairness, we compute *Jain's fairness index* [37] on 60 seconds time window intervals, and we plot the average, minimum, and maximum values across all intervals until the workload completes.

5.2 CARBYNE in Testbed Experiments

In this section, we compare CARBYNE to the state-of-the-art solutions across multiple benchmarks and metrics, evaluate its impact on workloads as a whole and on individual jobs, dissect the sources of improvements, and show that CARBYNE has small amortized overheads.

5.2.1 Performance vs. Efficiency vs. Fairness

The Offline Case Figure 4 depicts fairness, the average JCT, and cluster efficiency in our cluster experiments on the TPC-DS workload in the offline case.

We observe that DRF is the most fair approach, with a fairness index of 0.86 on average. However, CARBYNE is only 0.05 units away. In comparison, Tetris is off by 0.12 units and SJF by 0.22. CARBYNE can be unfair on small time intervals due to leftover reallocation, during which jobs can get more or less than their fair share. However, on longer time intervals (60 seconds in our experiments), it is closest to DRF because of its long-term approach toward fairness.

The average JCT is improved the most by SJF, and CARBYNE provides only 1.06× worse average JCT. In comparison, Tetris and DRF are off by 1.46× and 1.59× on average, respectively. Although CARBYNE performs SRTF during leftover reallocation, it is slightly worse than SJF because it attempts not to delay any job beyond its fair-share-calculated completion time.

Finally, Tetris provides the highest cluster efficiency (lowest makespan) by efficiently packing tasks across jobs, and CARBYNE is the closest scheme, which is only 1.03× worse than Tetris. Although CARBYNE is also packing tasks, it favors tasks from altruistic jobs instead of treating all runnable tasks equally. In comparison, DRF and SJF are off by 1.26× and 1.43×, respectively.

The Online Case Next, we focus on the case when jobs arrive over time (Section 5.1 has details on arrival process). Figure 5 shows that even in the online case, CARBYNE can closely match DRF (0.06 units worse), SJF (1.2× worse), and Tetris (1.07× worse) in fairness, performance, and efficiency, respectively – although the margins are slightly larger than that in the offline case.

Next, we investigate the root causes behind CARBYNE performing worse in the online case and whether they match our hypotheses in Section 3.4.

5.2.2 JCT Improvements Across Entire Workloads

While CARBYNE performs well for different metrics, the most important metric from a user's perspective is the average JCT. To better understand CARBYNE's impact on JCT, we compare it against popular alternatives: DRF and Tetris (§5.1). Note that we do not include SJF from hereon because it performs the worst both in terms of ef-

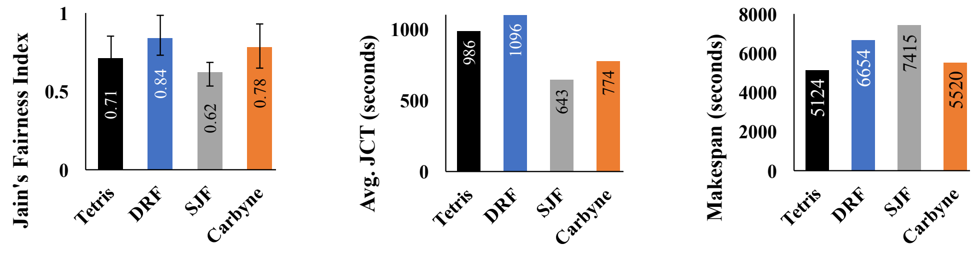


Figure 5: [Cluster] CARBYNE's performance against the best schemes in achieving fairness (DRF), improvement in average JCT (SJF), and cluster efficiency (Tetris) in the *online* case. CARBYNE approaches the best in each category and outperforms the rest.

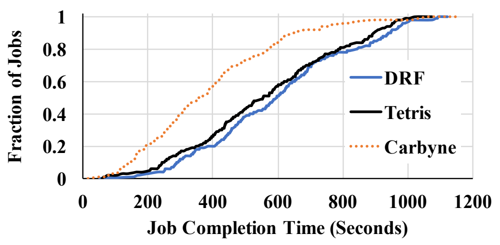


Figure 6: [Cluster] CDF of job completion times using different approaches on TPC-DS workload.

ficiency and fairness, while only marginally outperforming CARBYNE in performance.

Figure 6 shows the distributions of job completion times of the compared approaches for the TPC-DS workload. Only two highest percentiles are worse off by at most $1.1\times$ than Tetris using CARBYNE (not visible in Figure 6). Table 3 shows the corresponding improvement factors for multiple workloads.

CARBYNE vs. DRF For TPC-DS, CARBYNE speeds up jobs by $1.36\times$ on average and $1.88\times$ at the 95th percentile over DRF. Improvement factors are about the same for TPC-H and BigBench workloads. However, corresponding 75th and 95th percentile improvements were up to $1.62\times$ and $1.96\times$ for TPC-H. These gains are mostly due to the presence of a larger fraction of shorter jobs in the TPC-H workload, which benefit more due to CARBYNE's leftover allocation procedure.

CARBYNE vs. Tetris CARBYNE's improvements are similar against Tetris. Tetris ignores jobs dependencies and strives to pack, looking only at the current set of runnable tasks. In contrast, CARBYNE packs critical tasks during the leftover allocation.

5.2.3 Sources of Improvements

We have shown that workloads experience aggregate improvements using CARBYNE. A natural question is then to ask: where do these gains come from? To better answer that question, Figure 7 presents the factors of improvements of individual jobs.

	25th percentile		50th percentile	
Workload	**DRF**	**Tetris**	**DRF**	**Tetris**
TPC-DS	1.15	1.12	1.36	1.32
TPC-H	1.11	1.14	1.33	1.29
BigBench	1.13	1.10	1.41	1.35
	75th percentile		95th percentile	
Workload	**DRF**	**Tetris**	**DRF**	**Tetris**
TPC-DS	1.55	1.47	1.88	1.75
TPC-H	1.62	1.44	1.96	1.71
BigBench	1.57	1.52	1.85	1.82

Table 3: [Cluster] Factors of improvement across various workloads w.r.t. DRF and Tetris.

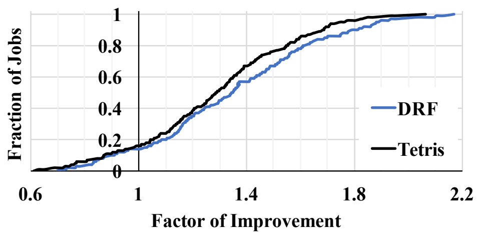


Figure 7: [Cluster] CDF of factors of improvement of individual jobs using CARBYNE w.r.t. different approaches.

We observe that for more than 84% of the jobs, CARBYNE performs significantly better than the alternatives. Only 16% of the jobs slow down – by at most $0.62\times$ – using CARBYNE w.r.t. different approaches. No more than 4% of the jobs slow down more than $0.8\times$.

Dissecting a Job's Execution To better understand how CARBYNE works in practice, we snapshot the execution of the same job from one of our TPC-DS cluster runs, with and without CARBYNE. Figure 8 presents the number of running tasks during the job execution when running CARBYNE and DRF. In both cases, the jobs were scheduled almost at the same time, approximately 300 seconds after our experiment has started.

The key takeaways are the following. First, in DRF, the breadth-first intra-job scheduler is greedily scheduling tasks whenever it has containers allocated (either due

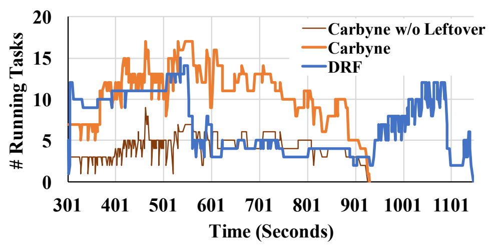


Figure 8: [Cluster] Snapshot of the execution of a TPC-DS query. The job switches from being altruistic in the earlier part of execution to receiving altruisms in the latter part, leading to faster completion. The gap between light and dark orange lines represent the tasks received from leftover allocation.

to fairness or work conservation mechanisms). However, being greedy does not necessarily help. Between times 570 and 940, its progress is slowed down mostly due to high contention in the cluster. Second, CARBYNE's directive of being altruistic helps in the long run. We observe that even when resources are available for allocation (interval between 300 and 550), its allocation is smoother than DRF (dark orange line). Instead of up to 11 tasks to be allocated, it decides to schedule up to 5 while giving up the remaining resources to the leftover. However, note that it does receive back some of the leftover resources (the gap between the light and dark orange lines in Figure 8). Finally, as the job nears completion, CARBYNE provides more leftover resources even when there is high contention in the cluster. In the interval between 570 seconds to 850 seconds, it is receiving significantly larger share than DRF by relying on other jobs' altruisms (i.e., the gap between the dark and light orange lines).

Which Jobs Slow Down? We observed that typically jobs with more work (many tasks and large resource demands per task) are more prone to losses, especially if they contend with many small jobs. One such example is a job that was slowed down by $0.64\times$ w.r.t DRF; it was contending with more than 40 jobs during its lifetime, all of them had less work to do, and hence, got favored by our leftover resource allocation policy.

However, unlike SJF or SRTF, CARBYNE is not inherently biased towards improving shorter jobs at the disadvantage of larger jobs. In fact, half of the jobs that slowed down more than $0.8\times$ (i.e., 2% of all jobs) were small.

We also attempted to bin/categorize the jobs based on characteristics such as the number of tasks, total amount of work, and DAG depth; however, but we did not observe any correlations w.r.t changes in JCT. Instead, we found that whether and how much a job may suffer is a function of the entire cluster's conditions during the job's runtime: available resources, rates of jobs arriving/finishing, their amount of remaining work, and the level of altruism. For better understand their impacts, we perform extensive simulations under multiple parameter choices in Section 5.4.

5.2.4 Scheduling Overheads

Recall from Section 4.2 that CARBYNE expands the Tez AM with an additional altruistic scheduling logic, which is triggered whenever the job's share allocation is changing. We find that CARBYNE-related changes inflate the decision logic by no more than 10 milliseconds, with a negligible increase in memory usage of the AM.

CARBYNE's logic to match tasks to machines – RM's matching logic happens on every heartbeat from NM to RM – is more complex than that in YARN. To quantify its overhead we compute the average time to process heartbeats from NM to RM for different number of pending tasks. We find that for 10000 pending tasks, CARBYNE's additional overhead is 2 milliseconds compared to Tetris (18 ms), and the overhead is up to 4 milliseconds for 50000 pending tasks. CARBYNE also extends the `Ask` requests from AM for multiple resource requirements and encapsulates `Asks` for the altruistic decisions it makes. However, because `Asks` are cumulative, we found that the additional overhead is negligible.

5.3 Performance in Trace-Driven Simulations

To better understand how CARBYNE performs under various parameter choices, we simulate altruistic scheduling using TPC-DS, TPC-H, and BigBench benchmarks as well as Microsoft and Facebook traces.

5.3.1 Benchmarks' Performance in Simulations

To evaluate the fidelity of our simulator, first we replayed the TPC-DS trace logs from cluster experiments in simulation. Table 4 shows the factors of improvement in JCT for TPC-DS workload in simulation that are consistent with that from our cluster experiments (Table 3). Similar results are obtained for the other workloads.

CARBYNE improves over the alternatives by up to $1.59\times$ on average and $7.67\times$ at the 95th percentile. Note that in simulation CARBYNE's improvements are slightly better than that in practice at higher percentiles. This is mainly due to natural factors such as failures, stragglers, and other delays that are not captured by our simulator.

5.3.2 Large-Scale Simulation on Production Trace

Table 4 also shows improvements for a 3000 (10000)-machine Facebook (Microsoft) production trace. CARBYNE outperforms others by up to $2.85\times$ ($8.88\times$) and $2.23\times$ ($7.86\times$) on average (95th percentile).

Note that the gains in production traces are significantly larger than that for other benchmarks. The primary reason is the increased opportunities for temporal rearrangements, both due to more jobs and more machines. In the Facebook trace, all the jobs are MapReduce jobs, and a significant fraction are Map-only jobs. Because

	25th percentile		50th percentile	
Workload	**DRF**	**Tetris**	**DRF**	**Tetris**
TPC-DS	1.17	1.12	1.59	1.47
Facebook	1.24	1.24	2.75	2.85
Microsoft	1.16	1.12	2.23	1.74
	75th percentile		95th percentile	
Workload	**DRF**	**Tetris**	**DRF**	**Tetris**
TPC-DS	2.73	2.23	7.67	6.17
Facebook	4.31	4.31	8.77	8.88
Microsoft	3.67	3.28	7.86	7.05

Table 4: [Simulation] Factors of improvement across various workloads w.r.t. DRF and Tetris.

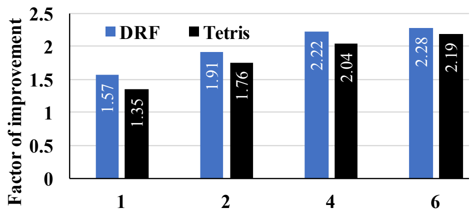


Figure 9: [Simulation] CARBYNE improvements over the alternatives for different cluster loads.

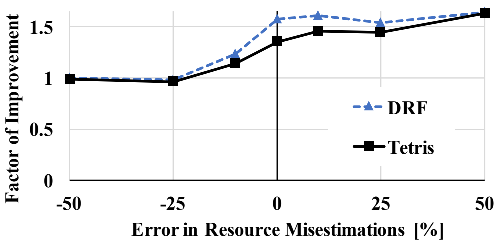


Figure 10: [Simulation] CARBYNE's improvements in terms of average JCT in the presence of misestimations in tasks' resource demands for different intra-job schedulers.

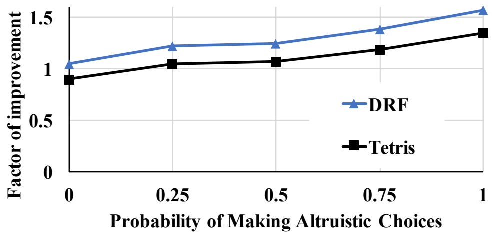


Figure 11: [Simulation] Benefits over the alternatives increase as CARBYNE makes altruistic choices more often. By default, CARBYNE uses 1; i.e., it is altruistic whenever possible.

these jobs only have one or two stages, tasks are less constrained. On the other hand, jobs in Microsoft trace are more complex DAGs with barriers which enables many opportunities for altruism. Furthermore, many jobs are large and have tens to thousands of tasks that can run in parallel. Together, they open up many opportunities for CARBYNE to use the leftover resources.

5.4 Sensitivity Analysis

5.4.1 Impact of Contention

In order to examine CARBYNE performance at different levels of contention, we vary load by changing the number of servers while keeping the workload constant. For example, half as many servers leads to twice as much load on the cluster. Figure 9 shows our results. At $1\times$ cluster load, CARBYNE improves over alternatives by up to $1.57\times$ on average. However, as we increase resource contention, CARBYNE's gains keep increasing. For example, at $2\times$ the load, its gains are between $1.76\times$ and $1.91\times$ on average. This is expected because the more the contention, the better is CARBYNE in carefully rearranging tasks over time. The performance gap increases even more at $4\times$ load. However, at $6\times$, CARBYNE's improvements are almost the same as that at $4\times$ load; this is because the cluster became saturated, without much room for leftover allocations.

5.4.2 Impact of Misestimations

CARBYNE assumes that we can accurately estimate tasks' resource demands and durations. However, accurate estimations can be challenging in practice, and misestimations are inevitable. To understand the impact of misestimations on CARBYNE, we introduce $X\%$ errors in our estimated demands and durations. Specifically, we select $X \in [-50, 50]$, and decrease/increase task resource demands by $\text{task}_{\text{newReq}} = (1 + X/100) * \text{task}_{\text{origReq}}$; task durations are changed accordingly.

Figure 10 shows CARBYNE's performance for varying degrees of estimation errors. It performs consistently better and becomes comparable when we underestimate, because underestimations encourage higher parallelism (due to false sense of abundance), disregarding altruistic scheduling. We also selectively introduced errors only to some of the tasks (not shown) with similar results: CARBYNE outperformed the rest despite misestimations.

5.4.3 Impact of Altruism Levels

The core idea behind CARBYNE is that jobs should altruistically give up resources that do not improve their JCT. In this experiment, we study how different levels of altruism ($P(Altruism)$) impact CARBYNE's benefits.

We observe in Figure 11 that increasing levels of altruism increases CARBYNE's advantage over the alternatives – CARBYNE outperforms them by up to $1.24\times$ when jobs are altruistic half the time ($P(Altruism) = 0.5$) and up to $1.56\times$ at $P(Altruism) = 1$. Last but not the least, at $P(Altruism) = 0$, CARBYNE is comparable to its alternatives; meaning, despite jobs being aggressive in their scheduling, our mechanism is still robust.

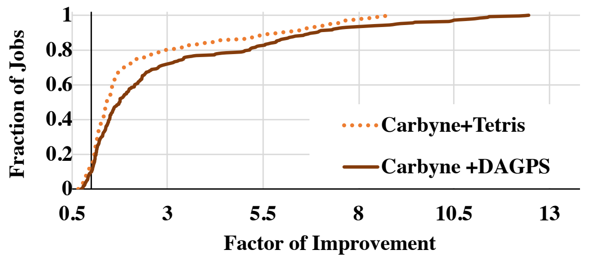


Figure 12: [Simulation] CDF of factors of improvement of individual jobs using CARBYNE + Tetris [29] and CARBYNE + DAGPS [30, 31] w.r.t. DRF. Benefits increase when CARBYNE uses a better DAG scheduler.

5.5 Impact of a Better DAG Scheduler

We also performed simulations where CARBYNE uses DAGPS [30, 31], a more sophisticated intra-job scheduler that considers the entire DAG and its dependencies between stages, instead of Tetris. Figure 12 shows that DAGPS can further increase CARBYNE's performance. We observe that for at least 50% of the jobs, CARBYNE + DAGPS performs better than CARBYNE + Tetris and factors of improvement increase from $2.36\times$ to $3.41\times$ for at least 25% of the jobs w.r.t. DRF. Similar observations hold when comparing with Tetris instead of DRF as the baseline. The additional gains are mainly due to DAGPS's ability to extract more leftover resources without affecting individual DAG completion times.

5.6 Comparison To Multi-Objective Schedulers

As mentioned in Section 1, prior solutions have also attempted to simultaneously meet multiple objectives. For example, Tetris [29] combines heuristics that improve cluster efficiency with those that lower average job completion time, and provides a knob (f) to trade-off fairness for performance. So far, we have shown CARBYNE's benefits against Tetris optimized for performance ($f \rightarrow 0$). Now we compare CARBYNE with Tetris offering strict fairness ($f \rightarrow 1$); we refer to this as $\text{Tetris}_{\text{Fair}}$. We found CARBYNE to be more fair than $\text{Tetris}_{\text{Fair}}$, with the average Jain's fairness index of 0.89 instead of 0.84. This is due to CARBYNE's ability to adhere strictly to fair allocations than $\text{Tetris}_{\text{Fair}}$'s fairness knob. CARBYNE also improves the average job completion time by up to $1.22\times$ and $2.9\times$ at the $95th$ percentile. Although CARBYNE's altruistic behavior to delay tasks in time play a significant role in getting these benefits, $\text{Tetris}_{\text{Fair}}$'s strategy to adopt strict fairness also limits the number of running jobs considered for efficient scheduling, which further hurts job completion times. Finally, these gains have direct implications on cluster efficiency, where CARBYNE outperforms $\text{Tetris}_{\text{Fair}}$ by $1.06\times$.

6 Related Work

Inter- and Intra-Job Schedulers Traditional cluster resource managers, e.g., Mesos [33] and YARN [51], employ inter-job schedulers [15, 21, 26, 27, 29, 35, 56] to optimize different objectives. For example, DRF [27] for fairness, shortest-job-first (SJF) [26] for minimizing the average JCT, and Tetris [29] for improving efficiency/utilization. More importantly, all of them focus on quick convergence to their preferred metric. Given some share of the resources, intra-job/task schedulers within each job optimize their own completion times. Examples include schedulers that process stages in a breadth-first order [2, 55], ones that follow the critical path [39, 40], and packers [29]. Again, all these schedulers are greedy in that they use up all resources allocated to them.

CARBYNE differs from all of them in at least two ways. First, CARBYNE takes an altruistic approach to maximize resources that can be redistributed. Second, by better redistributing the leftover resources, CARBYNE can simultaneously improve multiple objectives.

Altruistic Schedulers Delay scheduling for data locality [54] and coflow scheduling for network scheduling [20, 21] come the closest to CARBYNE in the high-level principle of altruism. The former waits to get better data locality, while the latter slows individual network flows down so that they all finish together. CARBYNE takes the next step by applying altruistic scheduling in the context of multi-resource scheduling. We leverage altruism to simultaneously improve multiple contrasting objectives.

Leftover Redistribution Hierarchical schedulers in both networking [50] and cluster computing [16] face the same problem as CARBYNE in terms of how to redistribute leftover resources. However, in those cases, entities do not voluntarily yield resources; they only yield resources after saturating their needs. Furthermore, they redistribute by fairly dividing resources among siblings in the hierarchy, whereas CARBYNE takes advantage of leftover resources to improve the average JCT and resource utilization without violating fairness.

DAG and Workflow Schedulers When the entire DAG with the completion times of all stages are known, the Critical Path Method (CPM) [39, 40] is one of the best known algorithms to minimize end-to-end completion times. However, it can be applied only as an intra-job scheduler. Many dynamic heuristics exist for online intra-DAG scheduling with varying results [53]. However, for multiple DAGS, i.e., for inter-DAG scheduling, existing solutions rely on either fair or shortest-first scheduling disciplines. In contrast, CARBYNE combines packing, ordering, and fair allocation.

7 Conclusion

Given the tradeoffs between fairness, performance, and efficiency, modern cluster schedulers [9, 10, 17, 33, 48, 51, 52] focus on performance isolation through *instantaneous* fairness and relegate performance and efficiency as best-effort, secondary goals. However, users perceive isolation only after jobs complete. As long as job completion times do not change, we can take a *long-term, altruistic* view instead of an instantaneous one. Using CARBYNE, jobs yield fractions of their resources without inflating their completion times. By combining and rescheduling these leftover resources from collective altruisms, CARBYNE significantly improves application-level performance and cluster utilization while providing the same level of performance isolation as modern schedulers. Deployments and large-scale simulations on benchmarks and production traces show that CARBYNE closely approximates DRF in terms of performance isolation, while providing 1.26× better efficiency and 1.59× lower average completion time.

Acknowledgments

We would like to thank the anonymous OSDI and SIGCOMM reviewers and our shepherd, Phil Levis, for their insightful comments and feedback that helped improve the paper. Robert and Aditya were supported in part by National Science Foundation (grants CNS-1302041, CNS-1330308, and CNS-1345249), Google and the Wisconsin Institute of Software-Defined Datacenters of Madison. Mosharaf was supported in part by National Science Foundation (grants CCF-1629397 and CNS-1563095) and Google.

References

[1] Apache Hadoop. http://hadoop.apache.org.

[2] Apache Tez. http://tez.apache.org.

[3] AWS Innovation at Scale. https://www.youtube.com/watch?v=JIQETrFC_SQ.

[4] Big-Data-Benchmark-for-Big-Bench. https://github.com/intel-hadoop/Big-Data-Benchmark-for-Big-Bench.

[5] Capacity Scheduler. https://hadoop.apache.org/docs/r2.4.1/hadoop-yarn/hadoop-yarn-site/CapacityScheduler.html.

[6] Chameleon. https://www.chameleoncloud.org/.

[7] TPC Benchmark DS (TPC-DS). http://www.tpc.org/tpcds.

[8] TPC Benchmark H (TPC-H). http://www.tpc.org/tpch.

[9] YARN Capacity Scheduler. http://goo.gl/cqwcp5.

[10] YARN Fair Scheduler. http://goo.gl/w5edEQ.

[11] S. Agarwal, S. Kandula, N. Burno, M.-C. Wu, I. Stoica, and J. Zhou. Re-optimizing data parallel computing. In *NSDI*, 2012.

[12] S. Agarwal, B. Mozafari, A. Panda, H. Milner, S. Madden, and I. Stoica. BlinkDB: Queries with bounded errors and bounded response times on very large data. In *EuroSys*, 2013.

[13] G. Ananthanarayanan, A. Ghodsi, S. Shenker, and I. Stoica. Effective Straggler Mitigation: Attack of the Clones. In *NSDI*, 2013.

[14] G. Ananthanarayanan, A. Ghodsi, A. Wang, D. Borthakur, S. Kandula, S. Shenker, and I. Stoica. PACMan: Coordinated memory caching for parallel jobs. In *NSDI*, 2012.

[15] G. Ananthanarayanan, S. Kandula, A. Greenberg, I. Stoica, Y. Lu, B. Saha, and E. Harris. Reining in the outliers in mapreduce clusters using Mantri. In *OSDI*, 2010.

[16] A. A. Bhattacharya, D. Culler, E. Friedman, A. Ghodsi, S. Shenker, and I. Stoica. Hierarchical scheduling for diverse datacenter workloads. In *SoCC*, 2013.

[17] E. Boutin, J. Ekanayake, W. Lin, B. Shi, J. Zhou, Z. Qian, M. Wu, and L. Zhou. Apollo: Scalable and coordinated scheduling for cloud-scale computing. In *OSDI*, 2014.

[18] M. Chowdhury, Z. Liu, A. Ghodsi, and I. Stoica. HUG: Multi-resource fairness for correlated and elastic demands. In *NSDI*, 2016.

[19] M. Chowdhury and I. Stoica. Efficient coflow scheduling without prior knowledge. In *SIGCOMM*, 2015.

[20] M. Chowdhury, M. Zaharia, J. Ma, M. I. Jordan, and I. Stoica. Managing data transfers in computer clusters with Orchestra. In *SIGCOMM*, 2011.

[21] M. Chowdhury, Y. Zhong, and I. Stoica. Efficient coflow scheduling with Varys. In *SIGCOMM*, 2014.

[22] E. G. Coffman and J. L. Bruno. *Computer and job-shop scheduling theory*. John Wiley & Sons, 1976.

[23] J. Dean and S. Ghemawat. MapReduce: Simplified data processing on large clusters. In *OSDI*, 2004.

[24] A. D. Ferguson, P. Bodik, S. Kandula, E. Boutin, and R. Fonseca. Jockey: Guaranteed job latency in data parallel clusters. In *Eurosys*, pages 99–112, 2012.

[25] M. Garey and D. Johnson. “Strong” NP-completeness results: Motivation, examples, and implications. *Journal of the ACM*, 25(3):499–508, 1978.

[26] M. R. Garey, D. S. Johnson, and R. Sethi. The complexity of flowshop and jobshop scheduling. *Mathematics of operations research*, 1(2):117–129, 1976.

[27] A. Ghodsi, M. Zaharia, B. Hindman, A. Konwinski, S. Shenker, and I. Stoica. Dominant Resource Fairness: Fair allocation of multiple resource types. In *NSDI*, 2011.

[28] A. Ghodsi, M. Zaharia, S. Shenker, and I. Stoica. Choosy: Max-min fair sharing for datacenter jobs with constraints. In *EuroSys*, 2013.

[29] R. Grandl, G. Ananthanarayanan, S. Kandula, S. Rao, and A. Akella. Multi-resource packing for cluster schedulers. In *SIGCOMM*, 2014.

[30] R. Grandl, S. Kandula, S. Rao, A. Akella, and J. Kulkarni. Do the hard stuff first: Scheduling dependent computations in data-analytics clusters. In *MSR-TR-2016-19*, 2016.

[31] R. Grandl, S. Kandula, S. Rao, A. Akella, and J. Kulkarni. Graphene: Packing and dependency-aware scheduling for data-parallel clusters. In *OSDI*, 2016.

[32] S. Gulwani, K. K. Mehra, and T. Chilimbi. SPEED: Precise and efficient static estimation of program computational complexity. In *POPL*, 2009.

[33] B. Hindman, A. Konwinski, M. Zaharia, A. Ghodsi, A. Joseph, R. Katz, S. Shenker, and I. Stoica. Mesos: A Platform for Fine-Grained Resource Sharing in the Data Center. In *NSDI*, 2011.

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

[35] M. Isard, V. Prabhakaran, J. Currey, U. Wieder, K. Talwar, and A. Goldberg. Quincy: Fair scheduling for distributed computing clusters. In *SOSP*, 2009.

[36] J. M. Jaffe. Bottleneck flow control. *IEEE Transactions on Communications*, 29(7):954–962, 1981.

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

[38] C. Joe-Wong, S. Sen, T. Lan, and M. Chiang. Multi-resource allocation: Fairness-efficiency tradeoffs in a unifying framework. In *INFOCOM*, 2012.

[39] J. E. Kelley. Critical-path planning and scheduling: Mathematical basis. *Operations Research*, 9(3):296–320, 1961.

[40] J. E. Kelley. The critical-path method: Resources planning and scheduling. *Industrial scheduling*, 13:347–365, 1963.

[41] M. Kornacker, A. Behm, V. Bittorf, T. Bobrovytsky, C. Ching, A. Choi, J. Erickson, M. Grund, D. Hecht, M. Jacobs, I. Joshi, L. Kuff, D. Kumar, A. Leblang, N. Li, I. Pandis, H. Robinson, D. Rorke, S. Rus, J. Russell, D. Tsirogiannis, S. Wanderman-Milne, and M. Yoder. Impala: A modern, open-source SQL engine for Hadoop. In *CIDR*, 2015.

[42] Y.-K. Kwok and I. Ahmad. Static scheduling algorithms for allocating directed task graphs to multiprocessors. *ACM Computing Surveys*, 31(4):406–471, 1999.

[43] Y. Low, J. Gonzalez, A. Kyrola, D. Bickson, C. Guestrin, and J. M. Hellerstein. GraphLab: A new framework for parallel machine learning. In *UAI*, 2010.

[44] K. Morton, M. Balazinska, and D. Grossman. ParaTimer: A progress indicator for MapReduce DAGs. In *SIGMOD*, 2010.

[45] D. G. Murray, F. McSherry, R. Isaacs, M. Isard, P. Barham, and M. Abadi. Naiad: A timely dataow system. In *SOSP*, 2013.

[46] D. C. Parkes, A. D. Procaccia, and N. Shah. Beyond Dominant Resource Fairness: Extensions, limitations, and indivisibilities. In *EC*, 2012.

[47] L. Popa, G. Kumar, M. Chowdhury, A. Krishnamurthy, S. Ratnasamy, and I. Stoica. FairCloud: Sharing the network in cloud computing. In *SIGCOMM*, 2012.

[48] M. Schwarzkopf, A. Konwinski, M. Abd-El-Malek, and J. Wilkes. Omega: Flexible, scalable schedulers for large compute clusters. In *EuroSys*, 2013.

[49] A. Singh, J. Ong, A. Agarwal, G. Anderson, A. Armistead, R. Bannon, S. Boving, G. Desai, B. Felderman, P. Germano, A. Kanagala, J. Provost, J. Simmons, E. Tanda, J. Wanderer, U. Hölzle, S. Stuart, and A. Vahdat. Jupiter rising: A decade of Clos topologies and centralized control in Google's datacenter network. In *SIGCOMM*, 2015.

[50] I. Stoica, H. Zhang, and T. Ng. A hierarchical fair service curve algorithm for link-sharing, real-time and priority service. In *SIGCOMM*, 1997.

[51] 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 *SoCC*, 2013.

[52] A. Verma, L. Pedrosa, M. Korupolu, D. Oppenheimer, E. Tune, and J. Wilkes. Large-scale cluster management at Google with Borg. In *EuroSys*, 2015.

[53] J. Yu, R. Buyya, and K. Ramamohanarao. Workflow scheduling algorithms for grid computing. In *Metaheuristics for Scheduling in Distributed Computing Environments*, pages 173–214. 2008.

[54] M. Zaharia, D. Borthakur, J. Sen Sarma, K. Elmeleegy, S. Shenker, and I. Stoica. Delay scheduling: A simple technique for achieving locality and fairness in cluster scheduling. In *EuroSys*, 2010.

[55] M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauley, M. Franklin, S. Shenker, and I. Stoica. Resilient Distributed Datasets: A fault-tolerant abstraction for in-memory cluster computing. In *NSDI*, 2012.

[56] M. Zaharia, A. Konwinski, A. D. Joseph, R. Katz, and I. Stoica. Improving MapReduce performance in heterogeneous environments. In *OSDI*, 2008.

Graphene: Packing and Dependency-aware Scheduling for Data-Parallel Clusters

Robert Grandl†+, Srikanth Kandula†, Sriram Rao†, Aditya Akella†+, Janardhan Kulkarni†
Microsoft† and University of Wisconsin-Madison+

Abstract– We present a new cluster scheduler, Graphene, aimed at jobs that have a complex dependency structure and heterogeneous resource demands. Relaxing either of these challenges, i.e., scheduling a DAG of homogeneous tasks or an independent set of heterogeneous tasks, leads to NP-hard problems. Reasonable heuristics exist for these simpler problems, but they perform poorly when scheduling heterogeneous DAGs. Our key insights are: (1) focus on the long-running tasks and those with tough-to-pack resource demands, (2) compute a DAG schedule, offline, by first scheduling such troublesome tasks and then scheduling the remaining tasks without violating dependencies. These offline schedules are distilled to a simple precedence order and are enforced by an online component that scales to many jobs. The online component also uses heuristics to compactly pack tasks and to trade-off fairness for faster job completion. Evaluation on a 200-server cluster and using traces of production DAGs at Microsoft, shows that Graphene improves median job completion time by 25% and cluster throughput by 30%.

1 Introduction

Heterogeneous DAGs are increasingly common in data-parallel clusters. We use DAG to refer to a directed acyclic graph where each vertex represents a task and edges encode input-output dependencies. Programming models such as Dryad, SparkSQL and Tez compile user scripts into job DAGs [2, 19, 24, 43, 57, 67]. Our study of a large production cluster in Microsoft shows that jobs have large and complex DAGs; the median DAG has a depth of five and thousands of tasks. Furthermore, there is a substantial variation in task durations (sub-second to hundreds of seconds) and the resource usage of tasks (e.g., compute, memory, network and disk bandwidth). In this paper, we consider the problem of scheduling such heterogeneous DAGs efficiently.

Given job DAGs and a cluster of machines, a cluster scheduler matches tasks to machines *online*. This matching has tight timing requirements due to the scale of modern clusters. Consequently, schedulers use simple heuristics. The heuristics leave gains on the table because they ignore crucial aspects of the problem. For example, critical path-based schedulers [36] only consider the critical path as determined by predicted task runtime and schedule tasks in the order of their critical path length. When DAGs have many parallel chains, running tasks that use different resources together can lead to a better schedule because it allows more tasks to run at the same time. As another example, multi-resource packers [37] aim to run the maximal number of pending tasks that fit within the available resources. When DAGs are deep, locally optimal choices do not always result in the fastest completion time of the whole DAG. Hence, intuitively, considering both variation in resource demands and dependencies may result in better schedules for heterogeneous DAGs.

By comparing the completion times of jobs in the production cluster with those achieved by an oracle, we estimate that the median job can be sped up by up to 50%. We observe that individual DAGs have fewer tasks running relative to the optimal schedule at some point in their lifetime. The cluster has lower overall utilization because (a) resources are idle even when tasks are pending due to dependencies or resource fragmentation, and (b) fewer jobs are released because users wait for the output of previous jobs. Given the large investment in such clusters, even a modest increase in utilization and job latency can have business impact [1, 10, 61].

We note that the optimal schedule for heterogeneous DAGs is intractable [54, 55]. Prior algorithmic work exists especially on simpler versions of the problem [18, 20, 21, 35, 50, 60, 65]. However, we are yet to find one that holds in the practical setting of a data-parallel cluster. Specifically, the solution has to work online, scale to large and complex DAGs as well as many concurrent jobs, cope with machine-level fragmentation as opposed to imagining one cluster-wide resource pool, and handle multiple objectives such as fairness, latency and throughput.

In this paper, we describe a cluster scheduler Graphene that efficiently schedules heterogeneous DAGs. To identify a good schedule for one DAG, we observe that the pathologically bad schedules in today's approaches mostly arise due to two reasons: (a) long-running tasks have no other work to overlap with them, which reduces parallelism, and (b) the tasks that are runnable do not

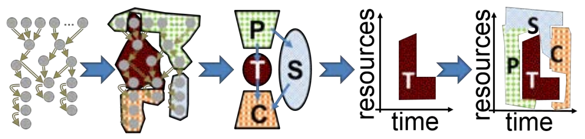


Figure 1: Steps taken by Graphene from a DAG on the left to its schedule on the right. Troublesome tasks T (in red) are placed first. The remaining tasks (parents P, children C and siblings S) are placed *on top* of T in a careful order to ensure compactness and respect dependencies.

pack well with each other, which increases resource fragmentation. Our approach is to identify the potentially *troublesome* tasks, such as those that run for a very long time or are hard to pack, and place them first onto a *virtual resource-time space*. This space would have $d+1$ dimensions when tasks require d resources; the last dimension being time. Our intuition is that placing the troublesome tasks first leads to a good schedule since the remaining tasks can be placed into resultant holes in this space.

At job submission time, Graphene builds a preferred schedule for a job as shown in Figure 1. After identifying a subset of troublesome tasks, the remaining tasks are divided into the parent, child and sibling subsets. Graphene first places the troublesome tasks onto a virtual resource-time space and then places the remaining subsets. Realizing this idea has a few challenges. Which choice of troublesome tasks leads to the best schedule? Further, since troublesome tasks are placed first, when a task is considered for placement some of its parent tasks and some of its children tasks may already have been placed in the virtual space. How to guarantee that every task can be placed without violating dependencies? Our answers are in §4.

Graphene's online component schedules the tasks of each DAG in the order of their starting time in the virtual resource-time space. Furthermore, across the many DAGs that may be running in the cluster, the online component respects different objectives– low job latency, high cluster throughput and fairness. These objectives can translate to discordant actions. For example, a fairness scheme such as DRF [33] may want to give resources to a certain job but the shortest-job-first heuristic that reduces job latency may pick a different job. Similarly, the task that is most likely to reduce resource fragmentation [37] may not start early in the virtual resource-time space. Our reconciliation heuristic intuitively picks tasks by consensus (e.g., based on a weighted combination of the scores received by a task from each objective). However, to maintain predictable performance, we limit unfairness to an operator-configured threshold.

We have implemented Graphene as extensions to Apache YARN and Tez and have experimented with jobs from TPC-DS, TPC-H and other benchmarks on a 200 server cluster. Furthermore, we evaluate Graphene in simulations on 20,000 DAGs from a production cluster.

To summarize, our key contributions are:

1. A characterization of the DAGs seen in production at Microsoft and an analysis of the performance of various DAG scheduling algorithms (§2).
2. A novel DAG scheduler that combines multi-resource packing and dependency awareness (§4).
3. An online inter-job scheduler that mimics the preferred per-job schedules while bounding unfairness (§5) for many fairness models [6, 33, 47].
4. In our extended tech report [38], we develop a new lower bound on the completion time of a DAG. Using this, we show that the schedules built by Graphene's offline component are within 1.04 times the theoretically optimal schedule (OPT) for half of the production DAGs; three quarters are within 1.13 times and the worst is 1.75 times OPT.
5. Our experiments show that Graphene improves the completion time of half of the DAGs by 19% to 31% across the various workloads. Production DAGs improve relatively more because those DAGs are more complex and have diverse resource demands. The gains accrue from running more tasks at a time; the cluster's job throughput (e.g., makespan) also improves by about 25%.

While we present our work in the context of cluster scheduling, DAGs are a powerful and general abstraction for scheduling problems. Scheduling the network transfers of a multi-way join or the work in a geo-distributed analytics job etc. can be represented as DAGs. We offer early results in §8 from applying Graphene to scheduling the DAGs that arise in distributed build systems [3, 34] and in request-response workflows [46, 66].

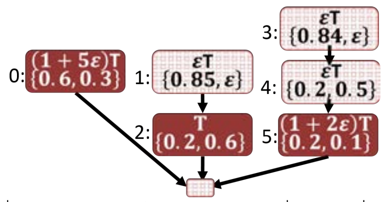
2 Primer on Scheduling Job DAGs

2.1 Problem definition

Let each job be represented as a directed acyclic graph $\mathcal{G} = \{V, E\}$. Each node in V is a task with demands for various resources. Edges in E encode precedence constraints between tasks. Many jobs can simultaneously run in a cluster. Given a set of concurrent jobs $\{\mathcal{G}\}$, the cluster scheduler maps tasks to machines while respecting capacity constraints and task dependencies. The goals of a typical cluster scheduler are *high performance* (measured using job throughput, average job completion time and overall cluster utilization) while offering *fairness* (measured w.r.t how resources are divided).

2.2 An illustrative example

We use the DAG shown in Figure 2 to illustrate the issues in scheduling DAGs. Each node represents a task: the node labels represent the duration (top) and the demands for two resources (bottom). Assume that the capacity is 1 for both resources. Let ε represent a value approaching zero.



0: $(1+5\varepsilon)T$ $\{0.6, 0.3\}$
1: εT $\{0.85, \varepsilon\}$
2: T $\{0.2, 0.6\}$
3: εT $\{0.84, \varepsilon\}$
4: εT $\{0.2, 0.5\}$
5: $(1+2\varepsilon)T$ $\{0.2, 0.1\}$

Technique	Execution Order	Time	Worst-case
OPT	$t_1 \to t_3 \to \{t_4, t_0\} \to \{t_0, t_2, t_5\}$	T	–
CPSched	$t_0 \to t_3 \to t_4 \to t_5 \to t_1 \to t_2$	$3T$	$O(n) \times$ OPT
Tetris	$t_0 \to t_1 \to t_3 \to t_2 \to t_4 \to t_5$	$3T$	$O(d) \times$ OPT

Figure 2: An example DAG where a packer (Tetris [37]) and a Critical Path scheduler take $3\times$ the optimal algo OPT. Here, Graphene is close to OPT (see §2.2). Assume $\varepsilon \to 0$.

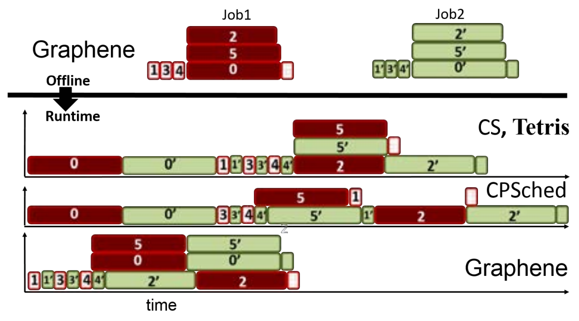


Figure 3: Illustrating the online case. Online, Graphene schedules the tasks of each DAG in the order of their start time in the offline schedule. Resources are to be divided fairly between two jobs both of which have the same DAG shown in Figure 2. Notice that fairness interleaves allocation between the jobs. Graphene's average job completion time and makespan improve by 25% to 75% and 50% to 100% respectively depending on the compared baseline.

Intuitively, a good schedule would overlap the long-running tasks shown with a dark background. The resulting optimal schedule (OPT) is listed in the table below the figure. OPT overlaps the execution of all the long-running tasks, t_0, t_2 and t_5, and finishes in T.

Since such long-running or resource-intensive tasks can be present anywhere in the DAG, greedy schedulers often perform poorly as we show next.

Critical path-based schedulers (CPSched) pick tasks along the critical path (CP) in the DAG. The CP for a task is the longest path from the task to the job output. The table shows the task execution order with CPSched.[1] CPSched ignores the resources needed by tasks. In this example, CPSched performs poorly because it does not schedule tasks off the critical path early (e.g., t_1, t_3, t_4) even though doing so reduces resource fragmentation by overlapping long-running tasks.

Packers, such as Tetris [37], match tasks to machines so as to maximize the number of simultaneously running tasks. Tetris greedily picks the task with the highest value of the dot product between task's demand vector and the available resource vector. The table also shows the task execution order with Tetris.[2] Tetris does not account for dependencies. Its packing heuristic only considers the tasks that are currently schedulable. In this example, Tetris performs poorly because it will not choose locally inferior packing options (such as running t_1 instead of t_0) even though that can lead to a better global packing.

Graphene comes close to the optimal schedule for this example. When searching for troublesome subsets, it will consider the subset $\{t_0, t_2, t_5\}$ because these tasks run for much longer. As shown in Figure 1, the troublesome tasks will be placed first. Since there are no dependencies among them, they will run at the same time. The parents ($\{t_1, t_3, t_4\}$) and any children are then placed *before* and *after* the troublesome tasks respectively in a compact manner while maintaining inter-task dependencies.

Online: Consider two jobs that have the DAG shown in Figure 2. Figure 3 illustrates the online schedule when resources are to be divided evenly between these jobs (e.g., slot fairness [13]).

The offline schedule computed by Graphene for each of the jobs, which overlaps the long-running tasks (t_0, t_2, t_5), is shown on top. The online component distills these schedules into a precedence order over tasks. For example, the order for both jobs is: $t_1, t_3, t_4, t_0, t_2, t_5$. Figure 3, bottom, shows a time-lapse of the task execution.

Capacity Scheduler (CS) [7], a widely used cluster scheduler, checks for which DAG the next available slot has to be allocated, and then picks (in a breadth-first order) a runnable task from the designated DAG that fits the available resources. Figure 3 shows that CS results in an average job completion time (JCT) and makespan of $2.5T$ and $3T$ respectively. Fairness causes the scheduler to interleave the tasks of the two jobs. Tetris happens to produce a similar schedule to CS. Note that this online schedule is far from the preferred per-DAG schedule, only a few of the long-running tasks overlap. Similar to CS, most production schedulers, including Spark, schedule tasks based on some topological ordering of the DAG while using fairness to decide which job to give resources to next. Hence, they behave similarly.

Figure 3 also shows that CPSched has an average JCT and makespan of $3.5T$ and $4T$ respectively. This is because CPSched finishes the t_1 tasks of both the jobs late; because t_1 has a small CP length. Therefore the t_2 tasks from both jobs do not overlap with any other long task.

Finally, the figure shows that Graphene has an average JCT and makespan of $2T$. Graphene achieves this by just enforcing a precedence order within each DAG. In particular, note that all of the schedulers are work-conserving; they leave resources idle only if no schedulable task can fit

[1]CP of t_0, t_1, t_3 is $T(1+5\varepsilon)$, $T(1+\varepsilon)$ and $T(1+4\varepsilon)$ respectively. Tasks can run simultaneously only if their total demand is below capacity.

[2]Tetris' packing score for each task, in descending order, is t_0=0.9, t_1=0.85, t_3=0.84, t_2=0.8, t_4=0.7 and t_5=0.3.

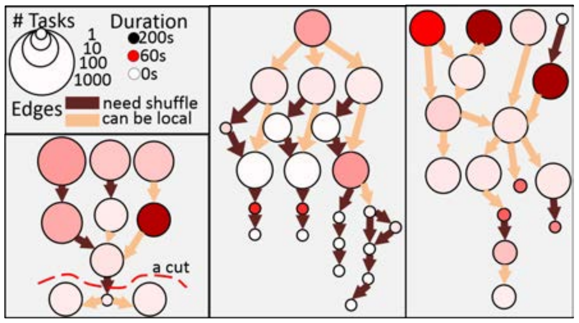


Figure 4: Visualizing a few small production DAGs. The legend is in the top left. See §2.3.

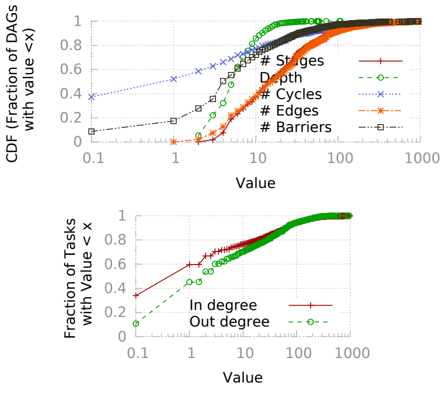


Figure 5: Characterizing structural properties of the DAGs

in those resources. The key difference, among the schedulers, is the *order* in which they consider the tasks for scheduling. Another difference is whether this *order* is computed based only on the runnable tasks (e.g., ordering runnable tasks on their CP length, packing score or on their breadth-first position) versus ordering based on a global optimization. Informally, GRAPHENE's gains arise from looking at the entire DAG and choosing a globally optimal schedule.

2.3 Analyzing DAGs in Production

To understand the problem with actual DAGs and at scale, we examine (a) the production jobs from a cluster of tens of thousands of servers at Microsoft, (b) jobs from a 200 server Hive [67] cluster and (c) jobs from a Condor cluster [5].

Structural properties: As a preliminary, Figure 4 illustrates some production DAGs at Microsoft. Each circle denotes a stage. By stage, we mean a collection of tasks that perform the same computation on different data (e.g. all map tasks). The size of the circle corresponds to the number of tasks in logarithmic scale, the circle's color corresponds to the average task duration in linear scale and the edge color denotes the type of dependency. We see W-shaped DAGs (bottom left) that join multiple datasets, inverted V-shaped DAGs (middle) that perform different analysis on a dataset, and more complex shapes (right) wherein multiple datasets are analyzed leading to multiple outputs. Note also the varying average durations of the tasks (circle colors); the resource variations are not shown for simplicity. Further, note cycles in the DAGs which are possibly due to self-joins and range-partitions.

Figure 5 plots a CDF of various structural properties of the DAGs from the Microsoft cluster. Since the x-axis is in log scale, we put the probability mass for $x = 0$ at $x = 0.1$.

We see that the median DAG has a depth of five. To compare, a map-reduce job has depth of one. A quarter of the DAGs have depth above ten.

While 40% of the DAGs are trees (i.e., no cycle after ignoring the direction of dependency), we see that many have cycles (half of the DAGs have at least 3 cycles); the average number of tasks in a cycle is 5 (not shown in the figure). Tree-like DAGs are an important special case as they are, theoretically, more tractable to schedule [48].

The DAGs can be quite large; the median job has thousands of tasks and tens of *stages*. To compare, a map-reduce job has two stages.

We also see that most of the edges are not barriers (e.g., those labeled "can be local"). Note the gap between the orange stars line and the black squares line in Figure 5 which correspond to counts of all edges and barriers respectively. A barrier edge indicates that every task in the parent stage should finish before any task in the child stage begins.

We observe that DAGs can be cut into portions such that all tasks after the cut can only begin after every task before the cut has finished. An example cut is shown with a red dashed line on the DAG in Figure 4 (left bottom). Cuts often arise because a dataset, perhaps newly generated by upstream tasks, has to be partitioned before downstream tasks can begin. Cuts are convenient because the optimal schedule for the DAG is a concatenation of the optimal schedules of the cut portions of that DAG. We observe that 24% of the production DAGs can be split into four or more parts.

The median (75th percentile) task in-degree and out-degree are 1 (8) and 3 (20) respectively. For a map-reduce job with m mappers and r reducers, the median in-degree will be 0 if $m \geq r$ and m otherwise. The larger out-degree is because stages that read from the file-system are data reductive; hence, the query optimizer creates fewer downstream tasks overall.

Overall, we conclude that DAGs are both large and have complex structures.

Diversity in resource demands: Similar to prior work [33, 37], we observed substantial variability in the usage of various resources; the details are in [38].

Potential for improvement: To quantify potential gains, we compare the runtime of production DAGs to two measures. The first, `CPLength`, is the duration of the DAG's critical path. If the available parallelism is infinite, the DAG would finish within `CPLength`. The second, `TWork`,

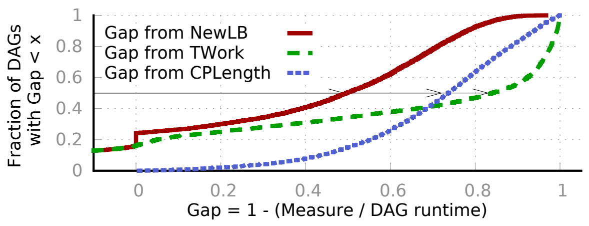


Figure 6: CDF of *gap* between DAG runtime and several measures. Gap is computed as $1 - \frac{\text{measure}}{\text{DAG runtime}}$.

is the total work in the DAG normalized by the cluster share of that DAG (a formula is in Table 1.) If there were no dependencies and perfect packing, a DAG would finish within `TWork`. Figure 6 plots a CDF of the relative gap between the runtime of a DAG and these measures. Half of the jobs have a gap of over 70% for both `CPLength` and `TWork`.

Understanding the gap: A careful reader would notice that about 15% of the DAGs finish faster than some measures. This is because our production scheduler occasionally gives jobs more than their fair share if the cluster has spare resources; hence, measures which assume that the cluster share will be the minimum guaranteed for the DAG can be larger than the actual completion time. We will ignore such DAGs for this analysis.

Suppose `OPT` is the optimal completion time for a DAG given a certain cluster share. We know that actual runtime is larger than `OPT` and that the above measures are smaller than `OPT`. Now, the gap could be due to one of two reasons. (1) The measure is loose (i.e., well below `OPT`). In practice, we found this to be the case because `CPLength` ignores all the work off the critical path and `TWork` ignores dependencies. (2) The observed runtimes of DAGs are inflated by runtime artifacts such as task failures, stragglers and performance interference from other cluster activity [17, 74].

To correct for (2), we discount the effects of runtime artifacts on the above computed DAG runtime as follows. First, we chose the fastest completion time from a group of recurring jobs. It is unlikely that every execution suffers from failures. Second, to correct for stragglers–one or a few tasks holding up job progress–we deduct from completion time the periods when the job ran fewer than ten concurrent tasks. Note that both these changes reduce the gap; hence they under-estimate the potential gain.

Further, to correct for (1), we develop a new improved lower bound `NewLB` that uses the specific structure of data-parallel DAGs. Further details are in [38]; but intuitively `NewLB` leverages the fact that all the tasks in a job stage (e.g., a map or reduce or join) have similar dependencies, durations and resource needs. The gap relative to `NewLB` is smaller, indicating that the newer bound is tighter, but the gap is still over 50% for half of the jobs. That is, they take over two times longer than they could.

To summarize, (1) production jobs have large DAGs that are neither a bunch of unrelated stages nor a chain of stages, and (2) a packing+dependency-aware scheduler can offer substantial improvements.

2.4 Analytical Results

Lemma 1 (Dependencies)**.** Any scheduling algorithm, deterministic or randomized, that does not account for the DAG structure (e.g., only schedules currently runnable tasks) is $\Omega(d)$ times `OPT` where d is the number of resources.

The proof, for deterministic algorithms, follows from designing an adversarial DAG for any scheduler [38]. We extend this to randomized algorithms by using Yao's minimax principle [38].

Lemma 1 applies to the following multi-resource packers [37, 58, 69, 70] since they ignore dependencies.

Lemma 2 (Resource Variation)**.** Schedulers that ignore resource heterogeneity have poor worst-case performance. For example, critical path scheduling can be $\Omega(n)$ times `OPT` where n is the number of tasks in a DAG.

The proof is by designing adversarial DAGs [38].

Combining these two principles, we conjecture that it is possible to find similar examples for any scheduler that ignores dependencies or ignores resource usages.

To place these results in context, note that d is about 4 (cores, memory, network, disk) and can be larger when tasks require resources at other servers or on many network links. Further, the median DAG has hundreds of tasks (n). The key intuition here is that DAGs are hard to schedule because of their complex structure and because of discretization issues when tasks use multiple resources (fragmentation, task placement etc.) GRAPHENE is close to `OPT` on all of the described examples and is within 1.04 times `OPT` for half of the production DAGs (see §7).

2.5 Acquiring annotated DAGs

Acquiring an annotated DAG is non-trivial. Much prior work has similar requirements as GRAPHENE (see Table 2 in [38]). There are two parts to this: the structure of the DAG and the task profiles (resource needs and durations).

DAG structure: In order to launch a task only after parent tasks finish, every DAG scheduler is aware of the DAG structure. Furthermore, the DAG is often known before the job starts. Runtime changes to the DAG, if they happen, only affect small portions of a DAG. For example, our scheduler adds an aggregation tree in front of a reduce stage depending upon runtime conditions.

Task resource demands and durations: GRAPHENE requires each task to be annotated with the demands for any resource that could be congested; the other resources do not affect scheduling. Here, we consider four resources (cores, memory, disk and network bandwidth).

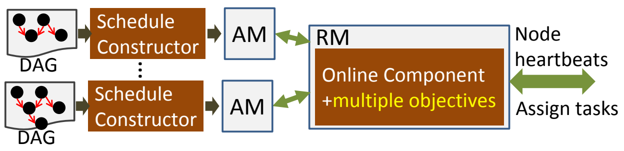


Figure 7: GRAPHENE builds schedules per DAG at job submission. The runtime component handles online aspects. AM and RM refer to the YARN's application and resource manager components.

Term	Definition
Task t	an atomic unit of execution
Stage s	a group of tasks that run same code on different data
`TWork`(s)	$\max_{\text{resource } r} \frac{1}{C_r} \sum_{t \in s} t_{\text{duration}} * t^r_{\text{demands}}$
`ExecTime`(s)	estimated time to execute tasks in s
$\mathcal{V}, \mathcal{G}$	$\mathcal{V}$ denotes all stages (and tasks) in a DAG $\mathcal{G}$
$\mathcal{S}$	a virtual schedule: i.e. a placement of a given DAG of tasks in a resource-time space
$C(s,\mathcal{G})$, $P(s,\mathcal{G})$, $D(s,\mathcal{G})$, $A(s,\mathcal{G})$, $U(s,\mathcal{G})$	Children, parents, descendants, ancestors and unordered neighbors of s in $\mathcal{G}$; note that $U(s,\mathcal{G}) = \mathcal{V} - A(s,\mathcal{G}) - D(s,\mathcal{G}) - \{s\}$

Table 1: Glossary of terms.

Schedulers such as Yarn, Mesos, Hive and Spark ask users to annotate their tasks with cores and memory requirements; for example, [1 core, 1 GB] is the default in Hadoop 2.6. GRAPHENE requires annotations for more resources as well as the durations of tasks.

There are some early efforts to obtain these profiles (tasks' demands and durations) automatically. For example, in the production cluster at Microsoft, up to 40% of the resources in the examined cluster are used by *recurring* jobs; the same script executes periodically on newly arriving data. Recurring jobs can be identified based on the job name (e.g., `LogMiner_date[_time]`) and prior work shows that the task profiles of these jobs can be estimated from history (after normalizing for the size of input) [16]. For the remaining jobs, some prior work builds profiles via sampling [59], program analysis [40], or based on online observations of the actual usages of tasks in the same *stage* [17]. Our method is described in §6; our sensitivity analysis in §7.4 shows that GRAPHENE is robust to modest amounts of estimation error.

3 Novel ideas in GRAPHENE

Cluster scheduling is the problem of matching tasks to machines. Most production schedulers today do so in an online manner and have very tight timing constraints since clusters have thousands of servers, many jobs that each have many pending tasks and tasks that finish in seconds or less [8, 73]. Given such stringent time budget, carefully considering large DAGs seems daunting.

As noted in §1, a key design decision in GRAPHENE is to divide this problem into two parts. An offline component constructs careful schedules for a single DAG. We call these the *preferred schedules*. A second online component enforces the preferred schedules of the various jobs running in the cluster. We elaborate on each of these parts below. Figure 7 shows an example of how the two parts may inter-operate in a YARN-style architecture. Dividing a complex problem into parts and independently solving each part often leads to a sub-optimal solution. While we have no guarantees for our particular division, we note that it scales to large clusters and outperforms the state-of-art in experiments.

To find a compact schedule for a single DAG, our idea is to place the troublesome tasks, i.e. those that can lead to a poor schedule, first onto a virtual space. Intuitively, this maximizes the likelihood that any *holes*, un-used parts of the resource-time space, can be filled by other tasks. However, finding the best choice of troublesome tasks is as hard as finding a good schedule for the DAG. We use an efficient search strategy that mimics dynamic programming: it picks subsets that are more likely to be useful and avoids redundant exploration. Furthermore, placing troublesome tasks first can lead to *dead-ends*. We define dead-end to be an arrangement of a subset of the DAG in the virtual space on which the remaining tasks cannot be placed without violating dependencies. Our strategy is to divide the DAG into subsets of tasks and place one subset at a time. While intra-subset dependencies are handled directly during schedule construction, inter-subset dependencies are handled by restricting the order in which the various subsets are placed. We prove that the resultant placement has no *dead-ends*.

The online component has to co-ordinate between some potentially discordant directives. Each job running in the cluster offers a preferred schedule for its tasks (constructed as above). Fairness models such as DRF may dictate which job (or queue) should be served next. The set of tasks that is advantageous for packing (e.g., maximal use of multiple resources) can be different from both the above choices. We offer a simple method to reconcile these various directives. Our idea is to compute a real-valued score for each pending task that incorporates the above aspects *softly*. That is, the score trades-off violations on some directives if the other directives weigh strongly against it. For example, we can pick a task that is less useful from a packing perspective if it appears much earlier on the preferred schedule. One key novel aspect is bounding the extent of unfairness.

The offline component of GRAPHENE is described next; the online component is described in Section 5.

4 Scheduling one DAG

GRAPHENE builds the schedule for a DAG in three steps. Figure 1 illustrates these steps and Figure 8 has a simplified pseudocode. First, GRAPHENE identifies some troublesome tasks and divides the DAG into four subsets (§4.1). Second, tasks in a subset are packed greedily onto the virtual space while respecting dependencies (§4.2). Third, GRAPHENE carefully restricts the order in which different

```
Func: BuildSchedule:
Input: $\mathcal{G}$: a DAG, $m$: number of machines
Output: An ordered list of tasks $t \in \mathcal{G}$
$\mathcal{S}_{best} \leftarrow \emptyset$ // best schedule for $\mathcal{G}$ thus far
foreach sets {T,S,P,C} ∈ CandidateTroublesomeTasks($\mathcal{G}$) do
    Space $\mathcal{S}$ ← CreateSpace($m$) //resource-time space
    $\mathcal{S}$ ← PlaceTasks(T, $\mathcal{S}$, $\mathcal{G}$) // trouble goes first
    $\mathcal{S}$ ← TrySubsetOrders({SCP, SPC, CSP, PSC}, $\mathcal{S}$, $\mathcal{G}$)
    if $\mathcal{S} < \mathcal{S}_{best}$ then $\mathcal{S}_{best} \leftarrow \mathcal{S}$ //keep the best schedule;
return OrderTasks($\mathcal{G}$, $\mathcal{S}_{best}$)
```

Figure 8: Pseudocode for constructing the schedule for a DAG. Helper methods are in Figure 9.

```
1  Func: CandidateTroublesomeTasks:
2  Input: DAG $\mathcal{G}$; Output: list $\mathcal{L}$ of sets T, S, P, C
   // choose a candidate set of troublesome tasks; per choice, divide $\mathcal{G}$
   into four sets
3  $\mathcal{L} \leftarrow \emptyset$
4  $\forall v \in \mathcal{G}$, LongScore($v$) ← $v$.duration/$\max_{v' \in \mathcal{G}} v'$.duration
5  $\forall v \in \mathcal{G}$, $v$ in stage $s$, FragScore($v$) ← TWork($s$)/ExecTime($s$)
6  foreach $l \in \delta, 2\delta, \ldots 1$ do
7      foreach $f \in \delta, 2\delta, \ldots 1$ do
8          T ← {$v \in \mathcal{G}$|LongScore($v$) ≥ $l$ or FragScore($v$) ≤ $f$}
9          T ← Closure(T)
10         if T ∈ $\mathcal{L}$ then continue // ignore duplicates;
11         P ← $\bigcup_{v \in T} A(v, \mathcal{G})$; C ← $\bigcup_{v \in T} D(v, \mathcal{G})$; S ← $\mathcal{V}$ − T − P − C;
12         $\mathcal{L} \leftarrow \mathcal{L} \cup$ {T, S, P, C}
```

Figure 9: Identifying various candidates for troublesome tasks and dividing the DAG into four subsets.

subsets are placed such that the troublesome tasks go first and there are no dead-ends (§4.3). Graphene picks the most compact schedule after iterating over many choices for troublesome tasks. The resulting schedule is passed on to the online component (§5).

4.1 Searching for troublesome tasks

To identify *troublesome* tasks, Graphene computes two scores per task. The first, `LongScore`, divides the task duration by the maximum across all tasks. Tasks with a higher score are more likely to be on the critical path and can benefit from being placed first because other work can overlap with them. The second, `FragScore`, reflects the packability of tasks in a stage (e.g., a map or a reduce). It is computed by dividing the total work in a stage (`TWork` defined in Table 1) by the time a greedy packer takes to schedule that stage. Tasks that are more difficult to pack would have a lower `FragScore`. Given thresholds l and f, Graphene picks tasks with `LongScore` $\geq l$ or `FragScore` $\leq f$. Intuitively, this biases towards selecting tasks that are more likely to hurt the schedule because they are long or difficult to pack. Each value of $\{l, f\}$ leads to a choice of troublesome tasks T which leads to a schedule (after placing the tasks in T first and then the other subsets); Graphene iterates over different values for the l and f thresholds and picks the most compact schedule.

To speed up this search, (1) rather than choose the threshold values arbitrarily, Graphene picks values that are discriminative, i.e. those that lead to different choices of troublesome tasks, and (2) Graphene remembers the set of troublesome tasks that were already explored (by previous settings of the thresholds) so that only one schedule is built for each troublesome set. Note also that the different choices of troublesome tasks can be explored in parallel. Further improvements are in §4.4.

As shown in Figure 9 (line 9), the set T is a closure over the chosen troublesome tasks. That is, T contains the troublesome tasks and all tasks that lie on a path in the DAG between two troublesome tasks. The parent and child subsets P, C consist of tasks that are not in T but have a descendant or ancestor in T respectively. The subset S consists of the remaining tasks.

4.2 Compactly placing tasks of a subset

Given a subset of tasks and a partially occupied space, how best to pack the tasks while respecting dependencies? Graphene uses the following logic for each of the subsets T, P, S and C. One can choose to place the parents first or the children first. We call these the *forward* and *backward* placements respectively. More formally, the forward placement recursively picks a task all of whose ancestors have already been placed on the space and puts it at the earliest possible time after its latest finishing ancestor. The backward placement is analogously defined. Intuitively, both placements respect dependencies but can lead to different schedules since greedy packing yields different results based on the order in which tasks are placed. Figure 10:PlaceTasks shows some simplified pseudo-code. Traversing the tasks in either placement has $n \log n$ complexity for a subset of n tasks and if there are m machines, placing tasks greedily has $n \log(mn)$ complexity.

4.3 Subset orders that guarantee feasibility

For each division of DAG into subsets T, S, P, C, Graphene considers these 4 orders: TSCP, TSPC, TPSC or TCSP. That is, in the TSCP order, it first places all tasks in T, then tasks in S, then tasks in C and finally all tasks in P. Intuitively, this helps because the troublesome tasks T are always placed first. Further, other orders may lead to dead-ends. For example, consider the order TPCS; by the time some task t in the subset S is considered for placement, parents of t and children of t may already have been placed since they may belong to the sets P and C respectively. Hence, it may be impossible to place t without violating dependencies. We prove that the above orders avoid *dead-ends* and are the only orders beginning with T to do so.

Note also that only one of the forwards or backwards placements (described in §4.2) are appropriate for some subsets of tasks. For example, tasks in P cannot be placed *forwards* since some descendants of these tasks may already have been placed (such as those in T). As noted above, the forwards placement places a task after its last finishing ancestor but ignores descendants and can hence violate dependencies if used for P; because by definition

```
Func: PlaceTasks($\mathcal{V}, \mathcal{S}, \mathcal{G}$):
Inputs: $\mathcal{V}$: subset of tasks to be placed, $\mathcal{S}$: space (partially filled), $\mathcal{G}$: a DAG
Output: a new space with tasks in $\mathcal{V}$ placed atop $\mathcal{S}$
return min (PlaceTasksF($\mathcal{V}, \mathcal{S}, \mathcal{G}$), PlaceTasksB($\mathcal{V}, \mathcal{S}, \mathcal{G}$))

Func: PlaceTasksF: // forwards placement, inputs and outputs same as PlaceTasks
$\mathcal{S} \leftarrow$ Clone($\mathcal{S}$)
finished placement set $F \leftarrow \{v \in \mathcal{G} | v$ already placed in $\mathcal{S}\}$
while true do
    ready set $R \leftarrow \{v \in \mathcal{V} - F \mid P(v, \mathcal{G}) \subseteq F\}$
    if $R = \emptyset$ then break // all done;
    $v' \leftarrow$ task in $R$ with longest runtime
    $t \leftarrow \max_{v \in P(v, \mathcal{G})}$ EndTime($v, \mathcal{S}$)
    // place $v'$ at earliest time $\geq t$ when its resource needs can be met
    $F \leftarrow F \cup v'$

Func: PlaceTasksB: // only backwards, analogous to PlaceTasksF.

Func: TrySubsetOrders:
Input: $\mathcal{G}$: a DAG, $\mathcal{S}_{\text{in}}$: space with tasks in T already placed
Output: A space that has the most compact placement of all tasks.
$\mathcal{S}_1, \mathcal{S}_2, \mathcal{S}_3, \mathcal{S}_4 \leftarrow$ Clone($\mathcal{S}_{\text{in}}$)
return min( // pick the most compact among all feasible orders
PlaceTasksF(C, PlaceTasksB(P, PlaceTasks(S, $\mathcal{S}_1$, $\mathcal{G}$), $\mathcal{G}$), $\mathcal{G}$), //SPC
PlaceTasksB(P, PlaceTasksF(C, PlaceTasks(S, $\mathcal{S}_2$, $\mathcal{G}$), $\mathcal{G}$), $\mathcal{G}$), //SCP
PlaceTasksB(P, PlaceTasksB(S, PlaceTasksF(C, $\mathcal{S}_3$, $\mathcal{G}$), $\mathcal{G}$), $\mathcal{G}$), //CSP
PlaceTasksF(C, PlaceTasksF(S, PlaceTasksB(P, $\mathcal{S}_4$, $\mathcal{G}$), $\mathcal{G}$), $\mathcal{G}$) //PSC
);
```

Figure 10: Pseudocode for the description in §4.2, §4.3.

every task in the parent subset P has at least one descendant task. Analogously, tasks in C cannot be placed *backwards*. Tasks in S can be placed in one or both placements, depending on the inter-subset order. Finally, since the tasks in T are placed onto an empty space they can be placed either forwards or backwards; details are in Figure 10:TrySubsetOrders. We prove the following:

Lemma 3. (Correctness) Our method in §4.1–§4.3 satisfies dependencies and avoids *dead-ends*. (Completeness) The method explores every order that places troublesome tasks first and is free of *dead-ends*.

Intuitively, the proof (omitted for space) follows from (1) all four subsets are closed and hence intra-subset dependencies are respected by the placement logic in §4.2 whether in the forward or in the backward placement, (2) the inter-subset orders and the corresponding restrictions to only use forwards and/or backwards placements specified in §4.3 ensure that dependencies across subsets are respected and, (3) every other order that begins with T can lead to dead-ends.

4.4 Enhancements

We note a few enhancements. First, as noted in §2.3, it is possible to partition a DAG into parts that are totally ordered. Hence, any schedule for the DAG is a concatenation of per-partition schedules. This lowers the complexity of schedule construction. 24% of the production DAGs can be split into four or more parts. Second, and along similar lines, whenever possible we reduce complexity by reasoning over stages. Stages are collections of tasks and are 10 to 10^3 times fewer in number than tasks. Third, schedule computation can be sped up in a few ways. Parallelizing the search will help the most, i.e. examine different choices for troublesome tasks T in parallel. Working over more compact representations (e.g., scaling down the DAG and the cluster by a corresponding amount) will also help. Fourth, jobs that are short-lived, or only use a small amount of resources, or do not have complex DAG structures, will bypass the offline portion of Graphene. Fifth, the complexity of schedule construction is independent of the sizes of the subsets T, S, P, C that Graphene divides the DAG into. However, if |T| is very large, the approach of placing troublesome tasks first and other tasks carefully around them is unlikely to help. We prune such choices of T without further exploration. Among the schedules built by Graphene for production DAGs, the median DAG has 17% of its tasks considered troublesome; these tasks contribute to 32% of the work in that job. Finally, note that it is possible to recursively employ this logic: i.e., given a DAG $\mathcal{G}$, pick a troublesome subset T, let $\mathcal{G}'$ be the sub-DAG over tasks in T, repeat the logic on $\mathcal{G}'$. We defer further examination of this approach to future work.

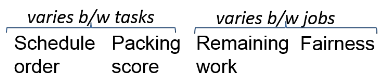


Figure 11: The various aspects considered by Graphene's online component when matching tasks to machines.

5 Scheduling many DAGs

Given the preferred schedules for each job, we describe how the Graphene inter-job scheduler matches tasks to machines online. Recall the example in Figure 3. The scheduling procedure is triggered when a machine m reports its vector of available resources to the cluster-wide resource manager. Given a set of runnable jobs (and their tasks), the scheduler returns a list of tasks to be allocated on that machine. The challenge is to enforce the per-job order computed in §4 while also packing tasks for cluster efficiency, ensuring low JCTs, and enforcing fairness.

5.1 Inter-job Scheduler

Enforcing preferred schedules. Using the per-DAG schedule constructed in §4, a $\texttt{t}_{priScore}$ is computed for each task t by (1) ranking tasks in increasing order of their start time in the schedule and (2) dividing the rank by the number of tasks in the DAG so that the result is between 1 (for the task that begins first) and 0. As noted below, Graphene preferentially schedules tasks with a higher $\texttt{t}_{priScore}$ value first.

Packing efficiency. Graphene borrows ideas from [37] to improve packing efficiency. For every task, it computes a packing score $\texttt{pScore}_t$ as a dot product between the task demand vector and the machine's available resource vector. To favor local placement, when remote resources are needed, $\texttt{pScore}_t$ is reduced by multiplying with a remote penalty rp ($\in [0,1]$). Sensitivity analysis on the value of rp is in §7.4.

Job completion time. Graphene estimates the remaining work in a job j similar to [37]; $\texttt{srpt}_j$ is a sum over the remaining tasks to schedule in j, the product of their duration and resource demands. A lower score implies less work remaining in the job j.

Bounding unfairness. Graphene trades off fairness for better performance while ensuring that the maximum unfairness is below an operator configured threshold. Specifically, Graphene maintains deficit counters [64] across jobs to measure unfairness. The deficit counters are updated as follows. When a task t from a group g is scheduled, its deficit increases by $\texttt{factor}_t \times (\texttt{fairShare}_g - 1)$ and the deficit of all the other groups g' increases by $\texttt{factor}_t \times \texttt{fairShare}_{g'}$. This update lowers the deficit counter of g proportional to the resources allocated to it and increases the deficit counters of other groups to remember that they were treated unfairly. Further, by varying the value of $\texttt{factor}_t$, Graphene can support different fairness schemes: e.g., $\texttt{factor}_t = 1$ mimics slot fairness and $\texttt{factor}_t =$ *demand of the dominant resource of g* mimics DRF [33].

Combining schedule order, packing, completion time and fairness. Graphene attempts to simultaneously consider the above four aspects; as shown in Figure 11, some of the aspects vary with the task while others vary across jobs. First, Graphene combines the performance related aspects into a single score, i.e., $\texttt{perfScore}_t = \text{pScore}_t \cdot \texttt{t}_{priScore} - \eta \texttt{srpt}_j$. η is a parameter that is automatically updated based on the average srpt and pScore across jobs and tasks. Subtracting $\eta \cdot \texttt{srpt}_j$ prefers shorter jobs. Sensitivity analysis on the value of η is in §7.4. Intuitively, the combined value $\texttt{perfScore}_t$ softly enforces the various objectives. For example, if a task t is preferred by all individual objectives (belongs to shortest job, is most packable, is next in the preferred schedule), then it will have the highest $\texttt{perfScore}_t$. When the objectives are discordant, colloquially, the task preferred by a majority of objectives t will have the highest $\texttt{perfScore}_t$.

Next, to trade-off performance while bounding unfairness, let the most unfairly treated group (the one with the highest deficit counter) be g_{unfair}. If the deficit counter of g_{unfair} is below the unfairness threshold, then Graphene picks the task with the maximum perfScore from among all groups; else it picks the task with the maximum perfScore from g_{unfair}. The unfairness threshold is κC where κ (< 1) is a tunable parameter and C is the cluster capacity.

Further details, including a pseudo-code, are in [38].

6 Graphene System

We have implemented the runtime component (§5) in the Apache YARN resource manager (RM) and the schedule constructor (§4) in the Apache Tez application master (AM). Our (unoptimized) schedule constructor finishes in tens of seconds on the DAGs used in experiments; this is in the same ballpark as the time to compile and query-optimize these DAGs. Recurring jobs use previously constructed schedules. Each DAG is managed by an instance of the Tez AM which closely resembles other frameworks such as FlumeJava [25] and Dryad [43]. The per-job AMs negotiate with the YARN RM for containers to run the job's tasks; each container is a fixed amount of various resources. As part of implementing Graphene, we expanded the interface between the AM and RM to pass additional information, such as the job's pending work and tasks' demands, duration and preferred order. Here, we describe some key aspects.

6.1 DAG Annotations

Recall from §2.5 that Graphene requires a more detailed annotation of DAGs than existing systems: specifically, it needs task durations and estimates of network and disk usages; the usages of cores and memory are already available [8, 67, 73].

Our approach is to construct estimates for the average task in each stage using a combination of historical data and prediction. These estimates are used by the offline portion of Graphene (§4). As noticed by prior work, recurring jobs are common in our production clusters and historical usages, after normalizing for the change in data volume, are predictive for such job groups [16]. The online portion of Graphene (§5) refines these estimates based on the actual work of a task (e.g., by noting its input size) and based on the executions of earlier tasks; since (a) tasks in the same stage often run in multiple waves due to capacity limits and (b) running tasks issue periodic progress reports [8, 17].

In our evaluation, we execute the jobs once and use the actual observed usage (from job history) to compute the necessary annotations. We normalize both the duration and usage estimates by the tasks' input size, as appropriate. A sensitivity analysis that introduces different amounts of error to the estimates and shows their effect on performance is in §7.4.

We observe that Graphene is rather robust to estimation error because relatively small differences in tasks' duration and usages do not change the schedule. For example, while it is useful to know that reduce and join tasks are network-heavy as opposed to map tasks which have no network usage, it is less useful to know precisely how much network usage a reducer or a join task will have; the actual usage would vary, at runtime, in any case due to contention, thread or process scheduling, etc. Similarly, while it is useful to know that tasks in a certain stage will take ten times longer, on average, and hence it is better to overlap those tasks with unrelated work, it is less useful to know the exact duration of a task; again, the exact durations will vary because of contention, machine-specific slowdowns etc. [17].

6.2 Efficient online matching

Naively implementing our runtime component (§5) would improve schedule quality at the cost of delaying scheduling. We use *bundling* to offset this issue.

Some background: The matching logic in typical schedulers is heartbeat-based [8]. When a machine heartbeats to the RM, the allocator (0) maintains an ordering over pending tasks, (1) picks the first appropriate task to allocate to that machine, (2) adjusts its data structures (such as, resorting/rescoring) and (3) repeats these steps until all resources on the node have been allocated or all allocation requests have been satisfied.

A naive implementation of the runtime component would examine all the pending tasks; thereby increasing the time to match.

Instead, we propose to bundle the allocations. Specifically, rather than breaking the loop after finding the first schedulable task (step 1 above), we keep along a *bundle* of tasks that can all be potentially scheduled on the machine. At the end of one pass, we assign multiple tasks by choosing from among those in the bundle.

The *bundle* amortizes the cost of examining the pending tasks. We can allocate multiple tasks in one pass as opposed to one pass per task. It is also easy to see that bundling admits non-greedy choices and that the pass can be terminated early when the bundle has good-enough tasks. We have refactored the Yarn scheduler with configurable choices for (1) which tasks to add to the bundle, (2) when to terminate bundling and (3) which tasks to pick from the bundle. From conversations with Hadoop committers, these code-changes help improve matching efficiency and code readability.

6.3 Co-existing with other features

We note that a cluster scheduler performs other roles besides matching tasks to machines. Several of these roles such as handling outliers and failed tasks differently [17, 74], delay scheduling [72], reservations [12, 30] or supporting heterogeneous clusters where only some servers may have GPUs [11] are implemented as preconditions to the main schedule loop, i.e. they are checked first, or are implemented by partitioning the tasks that will be considered in the scheduling loop. Since Graphene's changes only affect the inner core of the schedule loop (e.g., given a set of pending tasks, which subset to allocate to a machine), our implementation co-exists with these features.

7 Evaluation

Our key evaluation results are as follows.

(1) In experiments on a 200 server cluster, relative to `Tez` jobs running on `YARN`, Graphene improves completion time of half of the jobs by 19% to 31% across various benchmarks. 25% of the jobs improve by 30% to 49%. The extent of gains depends on the workload (complexity of DAGs, resource usage variations etc.).

(2) On over 20,000 DAGs from production clusters, the schedules constructed by Graphene are faster by 25% for half of the DAGs. A quarter of the DAGs improve by 57%. Further, by comparing with our new lower bound, these schedules are optimal for 40% of the jobs and within 13% of optimal for 75% of the jobs.

(3) By examining further details, we show that the gains are from better packing dependent tasks. Makespan (and cluster throughput) improve by a similar amount. More resources are used, on average, by Graphene and trading off short-term unfairness improves performance.

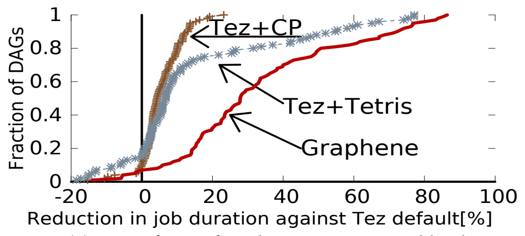
(4) We also compare with several alternative schedulers and offer a sensitivity analysis to cluster load, various parameter choices, and annotation errors.

7.1 Setup

Our experimental cluster has 200 servers with two quad-core Intel E2550 processors (hyperthreading enabled), 128 GB RAM, 10 drives, and a 10Gbps network interface. The network has a congestion-free core [39].

Workload: Our workload mix consists of jobs from public benchmarks—TPC-H [14], TPC-DS [15], Big-Bench [4], and jobs from a production cluster that runs Hive jobs (E-Hive). We also use 20K DAGs from a private production system in our simulations. In each experimental run, job arrival is modeled as a Poisson process with average inter-arrival time of 25s for 50 minutes. Each job is picked at random from the corresponding benchmark. We built representative inputs and varied input size from GBs to tens of TBs such that the average query completes in a few minutes and the longest query finishes in under ten minutes on the idle cluster. A typical experiment run has about 120 jobs. The results presented are the median over three runs.

Compared Schemes: We experimentally compare Graphene with the following baselines: (1) `Tez` : breadth-first order of tasks in the DAG running atop YARN's Capacity Scheduler (CS), (2) `Tez + CP` : critical path length based order of tasks in the DAG atop CS and (3) `Tez + Tetris` : breadth-first order of tasks in the DAG atop Tetris [37]. To tease apart the gains from the offline and online components, we also offer results for (4) `Tez + G + CS` and (5) `Tez + G + Tetris` which use the offline constructed schedules at the job manager (to request containers in that order) but the online components are agnostic to the desired schedule (either the default capacity scheduler or Tetris respectively). Using simulations, we also compare Graphene against the following schemes: (6) `BFS` : breadth first order, (7) `CP` : critical path order, (8) `Random` order, (9) `StripPart` [20], (10)



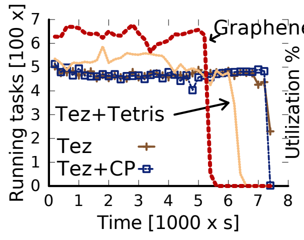
(a) CDF of gains for jobs on TPC-DS workload

Workload	50th percentile			75th percentile		
	G	T+C	T+T	G	T+C	T+T
TPC-DS	**27.8**	4.1	6.5	**45.7**	8.9	16.6
TPC-H	**30.5**	3.8	8.9	**48.3**	7.7	15.0
BigBench	**25.0**	6.4	6.2	**33.3**	21.7	18.5
E-Hive	**19.0**	1.0	5.8	**29.7**	4.5	14.2

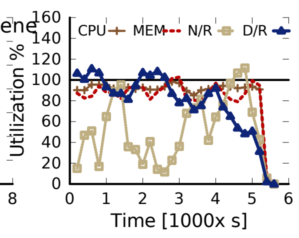
G stands for GRAPHENE. T+C and T+T denote `Tez + CP` and `Tez + Tetris` respectively (see §7.1). The improvements are relative to `Tez`.

(b) Improvements in JCT across all the workloads

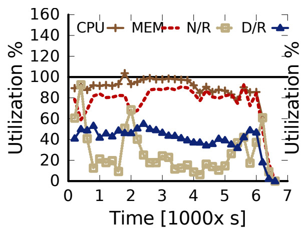
Figure 12: Comparing completion time improvements of various schemes relative to `Tez`.



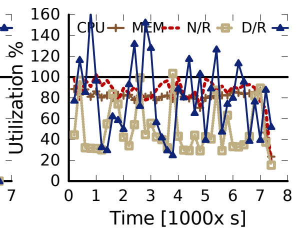
(a) Running tasks



(b) GRAPHENE



(c) `Tez + Tetris`



(d) `Tez + CP`

Figure 13: For a cluster run with 200 jobs, a time lapse of how many tasks are running (leftmost) and how many resources are allocated by each scheme. N/R represents the amount of network read, D/R the disk read and D/W the corresponding disk write.

`Tetris` [37], and (11) `CoffmanGraham` [29].

All of the above schemes except (9) are work-conserving. (6)–(8) and (10) pick only from among the runnable tasks but vary in the specific heuristic. (9) and (11) perform more complex schedule construction, as we will discuss later.

Metrics: Improvement in JCT is our key metric. Between two schemes, we measure the *normalized gap* in JCTs. That is, the difference in the runtime of a job divided by the job runtime; the normalization lets us compare jobs with very different runtimes. We also measure makespan, i.e., the time to finish a given set of jobs, Jain's fairness index [45], and the actual usages of various resources in the cluster.

7.2 How does GRAPHENE do in experiments?

Job Completion Time: Relative to `Tez`, Figure 12 shows that GRAPHENE improves half of the DAGs by 19 to 31%; the extent of gains depends on the workload and varies across benchmarks. A quarter of the DAGs improve by 30 to 49%. We see occasional regressions. Up to 5% of the jobs in the TPC-DS benchmark slow down with GRAPHENE; the maximum slowdown is 16%. We found this to primarily happen on the shorter jobs and believe it is due to noise from runtime artifacts such as stragglers and task failures [17]. The table in Figure 12 shows the results for all the benchmarks; we see that DAGs from E-Hive see the smallest improvement (19% at median) because the DAGs here are mostly two stage map-reduce jobs. The other benchmarks have more complex DAGs and hence receive larger gains.

Relative to the alternatives, Figure 12 shows that GRAPHENE is 15% to 34% better. `Tez + CP` achieves only marginal gains over `Tez`, hinting that critical path scheduling does not suffice. The exception is the BigBench dataset where about half the queries are dominated by work on the critical path. `Tez + Tetris` comes closest to GRAPHENE because Tetris' packing logic reduces fragmentation. The gap is still substantial since Tetris ignores dependencies. In fact, we see that `Tez + Tetris` does not consistently beat `Tez + CP`. Our takeaway is that considering both dependencies and packing substantially improves DAG completion time.

Where do the gains come from? Figure 13 offers more detail on an example experimental run. GRAPHENE keeps more tasks running on the cluster and hence finishes faster (Figure 13a). The other schemes take over 20% longer. GRAPHENE runs more tasks by reducing fragmentation and by overbooking resources such as network and disk that do not lose goodput when demand exceeds capacity (unlike say memory). Comparing Figure 13b with Figures 13c, 13d, the average allocation of all resources is higher with GRAPHENE. Occasionally, GRAPHENE allocates over 100% of the network and disk. One caveat about our measurement methodology here: we take the peak usage of a task and assume that the task held on to those resources for the entirety of its lifetime; hence, the usages are over-estimates for all schemes. `Tez + Tetris`, the closest alternative, has fewer tasks running at all times because (a) it does not overbook (resource usages are below 100% in Figure 13c) and (b) it has a worse global packing for a DAG because it ignores dependencies and packs only the runnable tasks. `Tez + CP` is impacted negatively by two effects: (a) ignoring disk and network usage leads to arbitrary over-allocation (the "total" resource usage is higher because, due to saturation, tasks hold on to allocations for longer) and (b) due to fragmentation, many fewer tasks run on average. Overall, GRAPHENE gains by increasing the task throughput.

Workload	Tez+CP	Tez+Tetris	GRAPHENE
TPC-DS	+2.1%	+8.2%	+30.9%
TPC-H	+4.3%	+9.6%	+27.5%

Table 2: Makespan, gap from Tez.

Workload	Scheme	2Q vs. 1Q Perf. Gap	Jain's fairness index		
			10s	60s	240s
TPC-DS	Tez	−13%	0.82	0.86	0.88
	Tez+DRF	−12%	0.85	0.89	0.90
	Tez+Tetris	−10%	0.77	0.81	0.92
	GRAPHENE	+2%	0.72	0.83	0.89

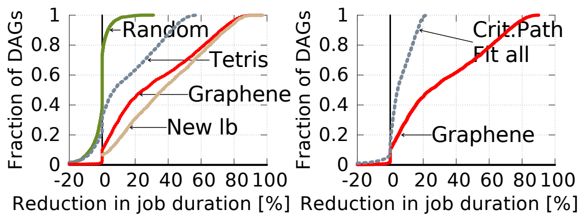
Table 3: Fairness: Shows the performance gap and Jain's fairness index when used with 2 queues (even share) versus 1 queue. Here, a score of 1 indicates perfect fairness.

Makespan: To evaluate makespan, we make one change to the experiment setup– all jobs arrive within the first few minutes. Everything else remains the same. Table 2 shows the gap in makespan for different cases. Due to careful packing, GRAPHENE sustains high cluster resource utilization which in turn enables jobs to finish quickly: makespan improves 30% relative to Tez and over 20% relative to alternatives.

Fairness: Can we improve performance while also being fair? Intuitively, fairness may hurt performance since fairly dividing resources may lower overall utilization or slow-down some jobs. To evaluate fairness, we make one change to the experiment set up. The jobs are evenly and randomly distributed among two queues and the scheduler has to divide resources evenly.

Table 3 reports the gap in performance (median JCT) for each scheme when run with two queues vs. one. Tez, Tez + DRF and Tez + Tetris lose over 10% in performance relative to their one queue counterparts. The table shows that with two queues, GRAPHENE has a small gain (perhaps due to experimental noise). Hence, relatively, GRAPHENE performs even better than the alternatives if given more queues. But why? Table 3 also shows Jain's fairness index computed over 10s, 60s and 240s windows. We see that GRAPHENE is less fair at short timescales but is indistinguishable at larger time windows. This is because GRAPHENE bounds unfairness (§5); it leverages short-term *slack* from precise fairness to make scheduling choices that improve performance.

Value of enforcing preferred schedules online: Recall that GRAPHENE's online component enforces the preferred schedules constructed by the offline component. To tease apart the value of this combination, we consider alternatives wherein the job managers use the preferred schedules (to request containers in that order) but the cluster scheduler is agnostic; i.e. it simply runs the default capacity scheduler or Tetris (we call these Tez + G + CS and Tez + G + Tetris respectively). We find that GRAPHENE offers 26% and 28% better median JCT compared to Tez + G + Tetris and Tez + G + CS. This experiment was conducted on a smaller 50 server cluster with different hardware so these numbers are not directly comparable with the remaining experiments; we offer them merely as



(a) GRAPHENE vs. Baselines (b) GRAPHENE vs. Alternates

Figure 14: Comparing GRAPHENE with other schemes. We removed the lines for CG and StripPart from the right figure because they hug $x = 0$; see Table 4.

		25th	50th	75th	90th
GRAPHENE		7	25	57	74
Random		−2	0	1	4
Crit.Path	Fit cpu/mem	−2	0	2	1
	Fit all	1	4	13	16
Tetris	Fit all	0	7	29	42
Strip Part.	Fit all	0	1	12	27
Coffman-Graham.	Fit all	0	1	12	26
	Fit cpu/mem	−2	0	0	2

Table 4: Reading out the gaps from Figure 14. Each entry is the improvement relative to BFS.

a qualitative validation of GRAPHENE's combination of online and offline components.

7.3 Comparing with alternatives

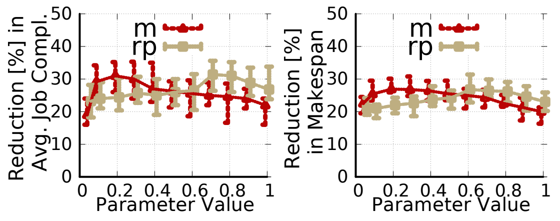
We use simulations to compare a wider set of algorithms (§7.1) on the much larger DAGs that ran in the production clusters. We mimic the actual dependencies, task durations and resource needs from the cluster.

Figure 14 compares the schedules constructed by GRAPHENE with the schedules from other algorithms. Table 4 reads out the gaps at various percentiles. We observe that GRAPHENE's gains at the end of schedule construction are about the same as those at runtime (Figure 12). This is interesting because the runtime component only softly enforces the desired schedules from all the jobs running simultaneously in the cluster. It appears that any loss in performance from not adhering to the desired schedule is made up by the gains from better packing (across DAGs) and trading off some short-term unfairness.

Second, GRAPHENE's gains are considerable compared to the alternatives. CP and Tetris are the closest. The reason is that GRAPHENE looks at the entire DAG and places the troublesome tasks first, leading to a more compact schedule overall.

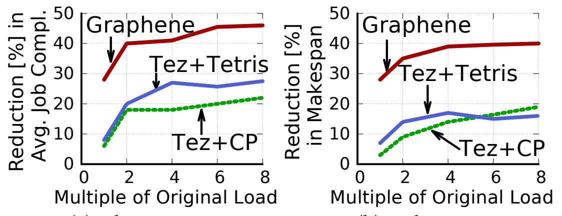
Third, when tasks have unit durations and nicely shaped demands, CG (Coffman-Graham [29]) is at most 2 times optimal. However, it does not perform well on the heterogeneous DAGs seen in production. Some recent extensions of CG to handle heterogeneity ignore fragmentation when resources are divided across machines [49].

Fourth, StripPart [20] combines resource packing and task dependencies and has the best-known approximation ratio: $O(\log n)$ on a DAG with n tasks [20]. The key idea is to partition tasks into *levels* such that all depen-



(a) Job Duration (b) Makespan

Figure 15: GRAPHENE - sensitivity analysis.



(a) Job Duration (b) Makespan

Figure 16: GRAPHENE's gains increase with cluster load.

dencies go across levels and then to tightly pack each level. We find that StripPart under-performs simpler heuristics in practice because (a) independent tasks that happen to fall into different levels cannot be packed together leading to wasted resources between levels and (b) the recommended per-level packers (e.g. [60]) do not support multiple resources and vector packing [58].

How close is GRAPHENE to Optimal? Comparing GRAPHENE with NewLB, we find that GRAPHENE is optimal for about 40% of the DAGs. For half (three quarters) of the DAGs, GRAPHENE is within 4% (13%) of the new lower bound. A gap still remains: for 10% of DAGs, GRAPHENE takes 25% longer. Manually examining these DAGs shows that NewLB is loose for most of them (deriving a tighter lower bound is an open problem). In sum, GRAPHENE is close to optimal for most of the production DAGs.

7.4 Sensitivity Analysis

Packing vs. Shortest Remaining Processing Time (srpt): Recall that we combine packing score and srpt using a weighted sum with η (§5). Let η be m times the average over the two expressions that it combines. Figure 15 shows the reduction in average JCT (on left) and makespan (on right) for different values of m. Values of $m \in [0.1, 0.3]$ have the most gains. Lower values lead to worse average JCT because the effect of srpt reduces; larger values lead to moderately worse makespan. Hence, we recommend $m = 0.2$.

Remote Penalty: GRAPHENE uses a remote penalty rp to prefer local placement. Our analysis shows that both JCT and makespan improve the most when rp $\in [0.7, 0.85]$ (Figure 15). Since rp is a multiplicative penalty, lower values of rp cause the scheduler to miss (non-local) scheduling opportunities whereas higher rp can over-use remote resources. We use rp $= 0.8$.

Cluster Load: We vary cluster load by reducing the number of available servers without changing the workload. Figure 16 shows the JCTs and makespan for a query set derived from TPC-DS. Both GRAPHENE and the alternatives offer more gains at higher loads. This is because the need for careful scheduling and packing increases when resources are scarce. Gains due to GRAPHENE increase by +10% at 2× load and by +15% at 6× load. Further, the gap between GRAPHENE and the alternatives remains similar across load levels.

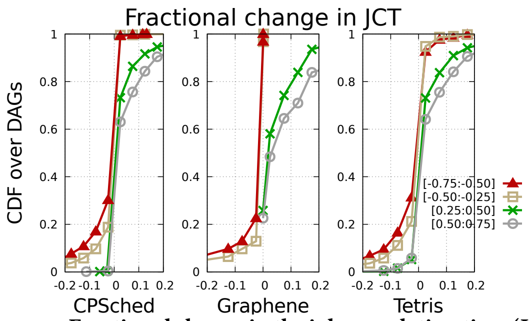


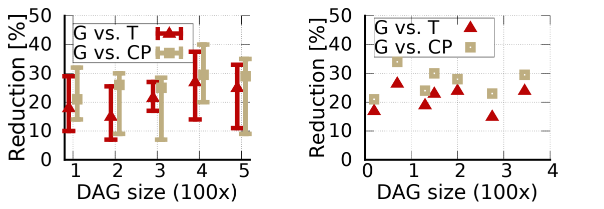
Figure 17: Fractional change in the job completion time (JCT) of DAGs with various schedulers when task durations and resource profiles are mis-estimated.

Impact of misestimations: We offer to each scheduler an inaccurate task duration and resource usage vector but have the underlying execution use the true values. Hence, the schedulers match tasks to machine based on imperfect estimates. Once scheduled, the tasks may finish after a different duration or use different amounts of resources. When the total resource demand crosses machine capacity, we delay the completion of tasks further by a proportional amount. Figure 17 shows a CDF of the change in the completion time of the production DAGs for different schedulers. Each line denotes a different amount of error. For example, the red triangle line labeled $[-0.75 : -0.50]$ corresponds to picking a random number in that range for each stage and then changing the task durations and resource needs fractionally by that random number (−0.75 indicates a 75% lower value). We see that the impact of mis-estimates is rather small; GRAPHENE changes roughly similarly to the other schedulers. Under-estimates tend to speed up the job because the scheduler over-allocates tasks but over-allocation can also slow-down jobs. Over-estimates delay jobs because the scheduler wastes resources; it may refrain from allocating a task when its needs appear larger than the available resources at a machine. Overall, GRAPHENE appears robust to mis-estimations.

8 Applying GRAPHENE to other domains

We evaluate GRAPHENE's effectiveness in scheduling DAGs that arise in distributed compilation jobs [3, 32, 34] and Internet service workflows [46].

Distributed build systems speed up the compilation of large code bases [3, 34]. Each build is a DAG with de-



(a) Distributed Build Systems: Compilation time (b) Request-response workflows: Query latency

Figure 18: Comparing Graphene (G) with Tetris (T) and Critical path scheduling (CP) on DAGs from other domains.

pendencies between the various tasks (compilation, linking, test, code analysis). The tasks have different runtimes and different resource profiles. Figure 18a shows that Graphene is 20% (30%) faster than `Tetris` (CP) when scheduling the build DAGs from a production distributed build system [32]. Each bar is centered on the median gain for DAGs of a certain size; the error bars are quartiles.

We also examine the DAGs that arise in datacenter-side workflows for Internet-services [46]. For instance, a search query translates into a workflow of dependent RPCs at the datacenter (e.g., spell check before index lookup, video and image lookup in parallel). The RPCs use different resources, have different runtimes and often run on the same server pool [46]. Over several workflows from a production service, Figure 18b shows that Graphene improves upon alternatives by about 24%. These encouraging early results hint that Graphene may be more broadly useful.

9 Related Work

To structure the discussion, we ask four questions: (Q1) does a scheme consider both packing and dependencies, (Q2) does it make realistic assumptions, (Q3) is it practical to implement in cluster schedulers and, (Q4) does it consider multiple objectives such as fairness? Graphene is unique in positively answering these four questions.

`Q1:NO`. Substantial prior work ignores dependencies but packs tasks with varying demands for multiple resources [26, 37, 60, 65, 71]. The best results are when the demand vectors are *small* [21]. Other work considers dependencies but assumes homogeneous demands [29, 36]. A recent multi-resource packing scheme, Tetris [37], succeeds on the three other questions but does not handle dependencies. Hence, we saw in §7 that Tetris performs poorly when scheduling DAGs (can be up to 2*d* times off, see [38]). Tetris can also be arbitrarily unfair.

`Q1:YES,Q2:NO`. The packing+dependencies problem has been considered at length under *job-shop scheduling* [31, 35, 50, 63]. Most results assume knowledge of job arrival times and profiles [49]. For the case with unknown future job arrivals (the version considered here), no algorithms with bounded competitive ratios are known [54, 55]. Some notable work assumes only two resources [23], applies for a chain but not a general DAG [18] or assumes one cluster-wide resource pool [51].

`Q3:NO`. Several of the schemes listed above are complex and hence do not meet the tight timing requirements of cluster schedulers. VM allocators [28] also consider multi-resource packing. However, cluster schedulers have to support roughly two to three orders of magnitude higher rate of allocation (tasks are more numerous than VMs).

`Q3:YES,Q1:NO`. Several works in cluster scheduling exist such as Quincy [44], Omega [62], Borg [68], Kubernetes [9] and Autopilot [42]. None of these combine multi-resource packing with DAG-awareness and many do neither. Job managers such as Tez [2] and Dryad [43] use simple heuristics such as breath-first scheduling and perform poorly in our experiments.

`Q4:NO`. Recently proposed fairness schemes incorporate multiple resources [33] and some are work-conserving [27]. We note that these fairness schemes neither pack nor are DAG-aware. Graphene can incorporate these fairness methods as one of the multiple objectives and trades off bounded unfairness for performance.

10 Concluding Remarks

DAGs are a common scheduling abstraction. However, we found that existing algorithms make key assumptions that do not hold in the case of cluster schedulers. Our scheduler, Graphene, is an efficient online solution that scales to large clusters. We experimentally validated that it substantially improves the scheduling of DAGs in both synthetic and emulated production traces. The core technical contributions are: (1) construct a good schedule for a DAG by placing tasks out-of-order on to a virtual resource-time space, and (2) use an online heuristic to softly enforce the desired schedules and simultaneously manage other concerns such as packing and fairness. Much of these innovations use the fact that job DAGs consist of groups of tasks (in each stage) that have similar durations, resource needs, and dependencies. We intend to contribute our Graphene implementation to Apache YARN/Tez projects.

Acknowledgments

For early discussions, we would like to thank Ganesh Ananthanaryanan and Peter Bodik. For feedback that helped improve this paper, we thank the anonymous reviewers, our shepherd Phil Levis, Mohammad Alizadeh, Chris Douglas, Hongzi Mao, Ishai Menache and Malte Schwarzkopf. For operating the cluster that motivated and inspired this work, we thank the Cosmos/ SCOPE production team at Microsoft.

References

[1] 43 bigdata platforms and bigdata analytics software. http://bit.ly/1DROqgt.

[2] Apache Tez. http://tez.apache.org/.

[3] Bazel. http://bazel.io/.

[4] Big-Data-Benchmark. http://bit.ly/1HlFRH0.

[5] Condor. http://research.cs.wisc.edu/htcondor/.

[6] Hadoop: Fair scheduler/ slot fairness. http://bit.ly/1PfsT7F.

[7] Hadoop MapReduce - Capacity Scheduler. http://bit.ly/1tGpbDN.

[8] Hadoop YARN Project. http://bit.ly/1iS8xvP.

[9] Kubernetes. http://kubernetes.io/.

[10] Market research on big-data-as-service offerings. http://bit.ly/1V6TXV4.

[11] Node labels in yarn. http://bit.ly/2d92ook.

[12] Reserved containers in yarn. http://bit.ly/2ckArD0.

[13] Slot and resource fairness in yarn. http://bit.ly/2ckArD0.

[14] TPC-H Benchmark. http://bit.ly/1KRK5gl.

[15] TPC-DS Benchmark. http://bit.ly/1J6uDap, 2012.

[16] Agarwal, S., Kandula, S., Burno, N., Wu, M.-C., Stoica, I., and Zhou, J. Re-optimizing data parallel computing. In *NSDI* (2012).

[17] Ananthanarayanan, G., Kandula, S., Greenberg, A., Stoica, I., Lu, Y., Saha, B., and Harris, E. Reining in the Outliers in MapReduce Clusters Using Mantri. In *OSDI* (2010).

[18] Anderson, E., Beyer, D., Chaudhuri, K., Kelly, T., Salazar, N., Santos, C., Swaminathan, R., Tarjan, R., Wiener, J., and Zhou, Y. Value-maximizing deadline scheduling and its application to animation rendering. In *SPAA* (2005).

[19] Armbrust, M., Xin, R. S., Lian, C., Huai, Y., Liu, D., Bradley, J. K., Meng, X., Kaftan, T., Franklin, M. J., Ghodsi, A., and Zaharia, M. Spark sql: Relational data processing in spark. In *SIGMOD* (2015).

[20] Augustine, J., Banerjee, S., and Irani, S. Strip packing with precedence constraints and strip packing with release times. In *SPAA* (2006).

[21] Azar, Y., Cohen, I. R., Fiat, A., and Roytman, A. Packing small vectors. In *SODA* (2016).

[22] Azar, Y., Kalp-Shaltiel, I., Lucier, B., Menache, I., Naor, J., and Yaniv, J. Truthful online scheduling with commitments. In *EC* (2015).

[23] Belkhale, K. P., and Banerjee, P. An approximate algorithm for the partitionable independent task scheduling problem. *Urbana 51* (1990), 61801.

[24] Chaiken, R., Jenkins, B., Larson, P.-A., Ramsey, B., Shakib, D., Weaver, S., and Zhou, J. SCOPE: Easy and Efficient Parallel Processing of Massive Datasets. In *VLDB* (2008).

[25] Chambers, C., Raniwala, A., Perry, F., Adams, S., Henry, R. R., Bradshaw, R., and Weizenbaum, N. Flumejava: Easy, efficient data-parallel pipelines. In *PLDI* (2010).

[26] Chandra, C., and Sanjeev, K. On multidimensional packing problems. *SIAM J. Comput.* (2004).

[27] Chowdhury, M., Liu, Z., Ghodsi, A., and Stoica, I. Hug: Multi-resource fairness for correlated and elastic demands. In *NSDI* (2016).

[28] Chowdhury, N., Rahman, M., and Boutaba, R. Virtual Network Embedding with Coordinated Node and Link Mapping. In *INFOCOM* (2009).

[29] Coffman, E.G., J., and Graham, R. Optimal scheduling for two-processor systems. *Acta Informatica* (1972).

[30] Curino, C., Difallah, D. E., Douglas, C., Krishnan, S., Ramakrishnan, R., and Rao, S. Reservation-based scheduling: If you're late don't blame us! In *SOCC* (2014).

[31] Czumaj, A., and Scheideler, C. A New Algorithm Approach to the General Lovasz Local Lemma with Applications to Scheduling and Satisfiability Problems (Extended Abstract). In *STOC* (2000).

[32] Esfahani, H., Fietz, J., Ke, Q., Kolomiets, A., Lan, E., Mavrinac, E., Schulte, W., Sanches, N., and Kandula, S. CloudBuild: Microsoft's Distributed and Caching Build Service. In *ICSE* (2016).

[33] Ghodsi, A., Zaharia, M., Hindman, B., Konwinski, A., Shenker, S., and Stoica, I. Dominant Resource Fairness: Fair Allocation Of Multiple Resource Types. In *NSDI* (2011).

[34] Gliboric, M., Schulte, W., Prasad, C., van Velzen, D., Narsamdya, I., and Livshits, B. Automated migration of build scripts using dynamic analysis and search-based refactoring. In *OOPSLA* (2014).

[35] Goldberg, L. A., Paterson, M., Srinivasan, A., and Sweedyk, E. Better approximation guarantees for job-shop scheduling. In *SODA* (1997).

[36] Graham, R. L. Bounds on multiprocessing timing anomalies. *SIAM Journal on Applied Mathematics* (1969).

[37] Grandl, R., Ananthanarayanan, G., Kandula, S., Rao, S., and Akella, A. Multi-resource Packing for Cluster Schedulers. In *SIGCOMM* (2014).

[38] Grandl, R., Kandula, S., Rao, S., Akella, A., and Kulkarni, J. Graphene: Packing and dependency-aware scheduling for data-parallel clusters, extended version. In *MSR Technical Report* (2016).

[39] Greenberg, A., 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 *SIGCOMM* (2009).

[40] Gulwani, S., Mehra, K., and Chilimbi, T. Speed: Precise and efficient static estimation of program computational complexity. In *POPL* (2009).

[41] Hindman, B., Konwinski, A., Zaharia, M., Ghodsi, A., Joseph, A. D., Katz, R., Shenker, S., and Stoica, I. Mesos: a platform for fine-grained resource sharing in the data center. In *NSDI* (2011).

[42] Isard, M. Autopilot: Automatic Data Center Management. *OSR* (2007).

[43] Isard, M., Budiu, M., Yu, Y., Birrell, A., and Fetterly, D. Dryad: Distributed Data-parallel Programs from Sequential Building Blocks. In *Eurosys* (2007).

[44] Isard, M., et al. Quincy: Fair Scheduling For Distributed Computing Clusters. In *SOSP* (2009).

[45] Jain, R., Chiu, D., and Hawe, W. A quantitative measure of fairness and discrimination for resource allocation in shared computer systems. *CoRR cs.NI/9809099* (1998).

[46] Jalaparti, V., Bodik, P., Kandula, S., Menache, I., Rybalkin, M., and Yan, C. Speeding up distributed request-response workflows. In *SIGCOMM* (2013).

[47] Kelly, F., Maulloo, A., and Tan, D. Rate control for communication networks: shadow prices, proportional fairness and stability. *Journal of the Operational Research Society* (1998).

[48] Kumar, V. S. A., Marathe, M. V., Parthasarathy, S., and Srinivasan, A. Scheduling on unrelated machines under tree-like precedence constraints. *Algorithmica* (2009).

[49] Kwok, Y., and Ahmad, I. Static scheduling algorithms for allocating directed task graphs to multiprocessors. *ACM Computing Surveys (CSUR)* (1999).

[50] Leighton, F. T., Maggs, B. M., and Rao, S. Universal packet routing algorithms. In *FOCS* (1988).

[51] Lepère, R., Trystram, D., and Woeginger, G. J. Approximation Algorithms for Scheduling Malleable Tasks Under Precedence Constraints. *International Journal of Foundations of Computer Science* (2002).

[52] Lucier, B., Menache, I., Naor, J., and Yaniv, J. Efficient online scheduling for deadline-sensitive batch computing. In *SPAA* (2013).

[53] Makarychev, K., and Panigrahi, D. Precedence-constrained scheduling of malleable jobs with preemption. In *ICALP* (2014).

[54] Mastrolilli, M., and Svensson, O. (acyclic) job shops are hard to approximate. In *FOCS* (2008).

[55] Monaldo, M., and Ola, S. Improved bounds for flow shop scheduling. In *ICALP* (2009).

[56] Motwani, R., and Raghavan, P. *Randomized Algorithms*. Cambridge University Press, New York, NY, USA, 1995.

[57] Olston, C., et al. Pig Latin: A Not-So-Foreign Language for Data Processing. In *SIGMOD* (2008).

[58] Panigrahy, R., Talwar, K., Uyeda, L., and Wieder, U. Heuristics for Vector Bin Packing. In *MSR TR* (2011).

[59] Peng, D., and Dabek, F. Large-scale incremental processing using distributed transactions and notifications. *DBMS 2006* (2010), 1–15.

[60] Schiermeyer, I. Reverse-fit: A2-optimal algorithm for packing rectangles. In *Proceedings of the Second Annual European Symposium on Algorithms* (1994).

[61] Schurman, E., and Brutlag, J. The User and Business Impact of Server Delays, Additional Bytes, and Http Chunking in Web Search.

http://velocityconf.com/velocity2009/public/schedule/detail/8523, 2009.

[62] Schwarzkopf, M., Konwinski, A., Abd-El-Malek, M., and Wilkes, J. Omega: Flexible, scalable schedulers for large compute clusters. In *EuroSys* (2013).

[63] Shmoys, D. B., Stein, C., and Wein, J. Improved approximation algorithms for shop scheduling problems. *SIAM J. Comput.* (1994).

[64] Shreedhar, M., and Varghese, G. Efficient fair queueing using deficit round robin. In *SIGCOMM* (1995).

[65] Sungjin, I., Nathaniel, K., Janardhan, K., and Debmalya, P. Tight bounds for online vector scheduling. In *FOCS* (2015).

[66] Suresh, L., Canini, M., Schmid, S., and Feldmann, A. C3: Cutting tail latency in cloud data stores via adaptive replica selection. In *NSDI* (2015).

[67] Thusoo, A., Sarma, J. S., Jain, N., Shao, Z., Chakka, P., Zhang, N., Antony, S., Liu, H., and Murthy, R. Hive- a warehousing solution over a map-reduce framework. In *VLDB* (2009).

[68] Verma, A., Pedrosa, L., Korupolu, M. R., Oppenheimer, D., Tune, E., and Wilkes, J. Large-scale cluster management at Google with Borg. In *EuroSys* (2015).

[69] Woeginger, G. J. There Is No Asymptotic PTAS For Two-Dimensional Vector Packing. In *Information Processing Letters* (1997).

[70] Yossi, A., Ilan, C., Seny, K., and Bruce, S. Tight bounds for online vector bin packing. In *STOC* (2013).

[71] Yossi, A., Ilan Reuven, C., and Iftah, G. The loss of serving in the dark. In *STOC* (2013).

[72] Zaharia, M., Borthakur, D., Sarma, J. S., Elmeleegy, K., Shenker, S., and Stoica, I. Delay Scheduling: A Technique For Achieving Locality And Fairness In Cluster Scheduling. In *EuroSys* (2010).

[73] Zaharia, M., Chowdhury, N. M. M., Franklin, M., Shenker, S., and Stoica, I. Spark: Cluster computing with working sets. No. UCB/EECS-2010-53.

[74] Zaharia, M., Konwinski, A., Joseph, A. D., Katz, R., and Stoica, I. Improving MapReduce Performance in Heterogeneous Environments. In *OSDI* (2008).

Firmament: fast, centralized cluster scheduling at scale

Ionel Gog[†] Malte Schwarzkopf[‡] Adam Gleave[†] Robert N. M. Watson[†] Steven Hand[*]
[†] *University of Cambridge Computer Laboratory* [‡] *MIT CSAIL* [*] *Google, Inc.*

Abstract

Centralized datacenter schedulers can make high-quality placement decisions when scheduling tasks in a cluster. Today, however, high-quality placements come at the cost of high latency at scale, which degrades response time for interactive tasks and reduces cluster utilization.

This paper describes Firmament, a centralized scheduler that scales to over ten thousand machines at sub-second placement latency even though it continuously reschedules all tasks via a min-cost max-flow (MCMF) optimization. Firmament achieves low latency by using multiple MCMF algorithms, by solving the problem incrementally, and via problem-specific optimizations.

Experiments with a Google workload trace from a 12,500-machine cluster show that Firmament improves placement latency by 20× over Quincy [22], a prior centralized scheduler using the same MCMF optimization. Moreover, even though Firmament is centralized, it matches the placement latency of distributed schedulers for workloads of short tasks. Finally, Firmament exceeds the placement quality of four widely-used centralized and distributed schedulers on a real-world cluster, and hence improves batch task response time by 6×.

1 Introduction

Many applications today run on large datacenter clusters [3]. These clusters are shared by applications of many organizations and users [6; 21; 35]. Users execute *jobs*, which each consist of one or more parallel *tasks*. The *cluster scheduler* decides how to place these tasks on cluster machines, where they are instantiated as processes, containers, or VMs.

Better task placements by the cluster scheduler lead to higher machine utilization [35], shorter batch job runtime, improved load balancing, more predictable application performance [12; 36], and increased fault tolerance [32]. Achieving high task *placement quality* is hard: it requires algorithmically complex optimization in multiple dimensions. This goal conflicts with the need for a low *placement latency*, the time it takes the scheduler to place a new task. A low placement latency is required both to meet user expectations and to avoid idle cluster resources while there are waiting tasks. Shorter batch task runtimes and increasing cluster scale make it difficult to meet both conflicting goals [9; 10; 13; 23; 29]. Current schedulers thus choose one to prioritize.

Three different cluster scheduler architectures exist today. First, *centralized* schedulers use elaborate algorithms to find high-quality placements [11; 12; 35], but have latencies of seconds or minutes [13; 32]. Second, *distributed* schedulers use simple algorithms that allow for high throughput, low latency parallel task placement at scale [13; 28; 29]. However, their uncoordinated decisions based on partial, stale state can result in poor placements. Third, *hybrid* schedulers split the workload across a centralized and a distributed component. They use sophisticated algorithms for long-running tasks, but rely on distributed placement for short tasks [9; 10; 23].

In this paper, we show that a centralized scheduler based on sophisticated algorithms can be fast and scalable for both current and future workloads. We built Firmament, a centralized scheduler that meets three goals:

1. to maintain the same high placement quality as an existing centralized scheduler (*viz.* Quincy [22]);
2. to achieve sub-second task placement latency for all workloads in the common case; and
3. to cope well with demanding situations such as cluster oversubscription or large incoming jobs.

Our key insight is that even centralized sophisticated algorithms for the scheduling problem can be fast (*i*) if they match the problem structure well, and (*ii*) if few changes to cluster state occur while the algorithm runs.

Firmament generalizes Quincy [22], which represents the scheduling problem as a min-cost max-flow (MCMF) optimization over a graph (§3) and continuously reschedules the entire workload. Quincy's original MCMF algorithm results in task placement latencies of minutes on a large cluster. Firmament, however, achieves placement latencies of hundreds of milliseconds in the common case and reaches the same placement quality as Quincy.

To achieve this, we studied several MCMF optimization algorithms and their performance (§4). Surprisingly, we found that *relaxation* [4], a seemingly inefficient MCMF algorithm, outperforms other algorithms on the graphs generated by the scheduling problem. However, relaxation can be slow in crucial edge cases, and we thus investigated three techniques to reduce Firmament's placement latency across different algorithms (§5):

1. Terminating the MCMF algorithms early to find *approximate solutions* generates unacceptably poor and volatile placements, and we reject the idea.
2. *Incremental re-optimization* improves the runtime of Quincy's original MCMF algorithm (cost scaling [17]), and makes it an acceptable fallback.
3. *Problem-specific heuristics* aid some MCMF algorithms to run faster on graphs of specific structure.

We combined these algorithmic insights with several implementation-level techniques to further reduce Firmament's placement latency (§6). Firmament runs two MCMF algorithms concurrently to avoid slowdown in edge cases; it implements an efficient graph update algorithm to handle cluster state changes; and it quickly extracts task placements from the computed optimal flow.

Our evaluation compares Firmament to existing distributed and centralized schedulers, both in simulation (using a Google workload trace) and on a local 40-machine cluster (§7). In our experiments, we find that Firmament scales well: even with 12,500 machines and 150,000 live tasks eligible for rescheduling, Firmament makes sub-second placements. This task placement latency is comparable to those of distributed schedulers, even though Firmament is centralized. When scheduling workloads that consist *exclusively* of short, sub-second tasks, Firmament scales to over 1,000 machines, but suffers overheads for task runtimes below 5s at 10,000 machines. Yet, we find that Firmament copes well with realistic, mixed workloads that combine long-running services and short tasks even at this scale: Firmament keeps up with a 250× accelerated Google workload. Finally, we show that Firmament's improved placement quality reduces short batch tasks' runtime by up to 6× compared to other schedulers on a real-world cluster.

Firmament is available as open-source software (§9).

2 Background

Cluster managers such as Mesos [21], YARN [34], Borg [35], and Kubernetes [14] automatically share and manage physical datacenter resources. Each one has a *scheduler*, which is responsible for placing tasks on machines. Figure 1 illustrates the lifecycle of a task in a cluster manager: after the user submits the task, it waits until the scheduler places it on a machine where it subsequently runs. The time between submission and task placement is the *task placement latency*, and to the total time between the task's submission and its completion is the *task response time*.[1] The time a task spends being actively scheduled is the scheduler's *algorithm runtime*.

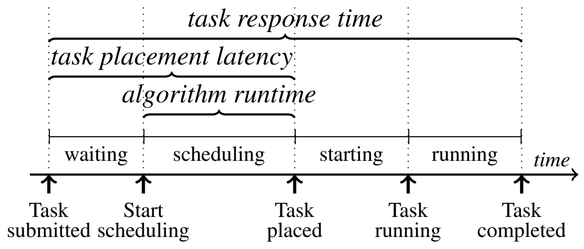


Figure 1: Task lifecycle phases, state transition events *(bottom)* and the time ranges used in this paper *(top)*.

For each task, the scheduling algorithm typically first performs a *feasibility check* to identify suitable machines, then *scores* them according to a preference order, and finally *places* the task on the best-scoring machine. Scoring, i.e., rating the different placement choices for a task, can be expensive. Google's Borg, for example, relies on several batching, caching, and approximation optimizations to keep scoring tractable [35, §3.4].

High placement quality increases cluster utilization and avoids performance degradation due to overcommit. Poor placement quality, by contrast, increases task response time (for batch tasks), or decreases application-level performance (for long-running services).

2.1 Task-by-task placement

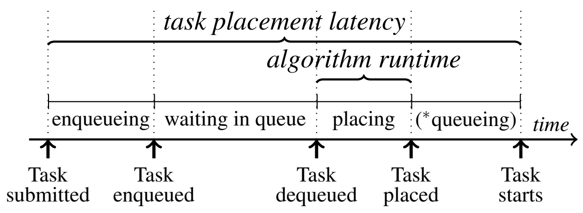
Most cluster schedulers, whether centralized or distributed, are *queue-based* and process one task at a time (per scheduler). Figure 2a illustrates how such a queue-based scheduler processes a new task. The task first waits in a queue of unscheduled tasks until it is dequeued and processed by the scheduler. In a busy cluster, a task may spend substantial time enqueued. Some schedulers also have tasks wait in a per-machine "worker-side" queue [29], which allows for pipelined parallelism.

Task-by-task placement has the advantage of being amenable to uncoordinated, parallel decisions in distributed schedulers [9; 10; 13; 28]. On the other hand, processing one task at a time also has two crucial downsides: first, the scheduler commits to a placement early and restricts its choices for further waiting tasks, and second, there is limited opportunity to amortize work.

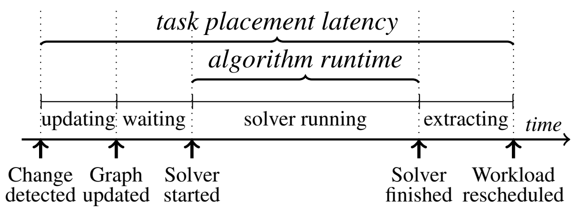
2.2 Batching placement

Both downsides of task-by-task placement can be addressed by batching. Processing several tasks in a batch

[1]Task response time is primarily meaningful for batch tasks; long-running service tasks' response times are conceptually infinite, and in practice are determined by failures and operational decisions.



(a) Queue-based schedulers (*: optional worker-side queue).



(b) Flow-based schedulers (Quincy [22], Firmament).

Figure 2: Tasks wait to be placed individually in queue-based schedulers (a), while flow-based schedulers (b) reschedule the whole workload in a long solver run, which makes it essential to minimize algorithm runtime at scale.

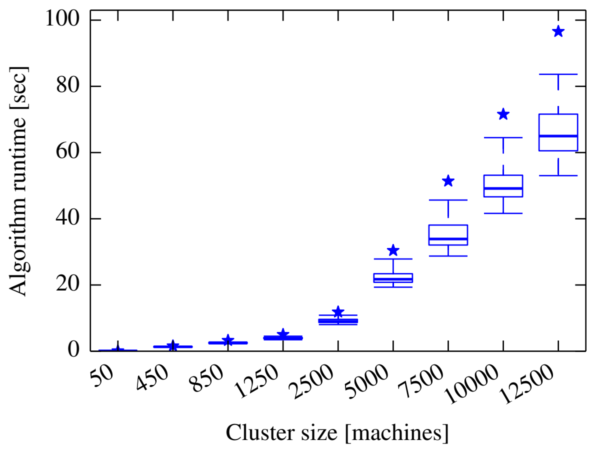


Figure 3: Quincy [22]'s approach scales poorly as cluster size grows. Simulation on subsets of the Google trace; boxes are 25th, 50th, and 75th percentile delays, whiskers 1st and 99th, and a star indicates the maximum.

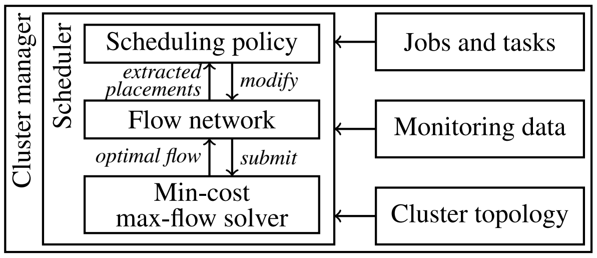


Figure 4: Firmament's scheduling policy modifies the flow network according to workload, cluster, and monitoring data; the network is passed to the MCMF solver, whose computed optimal flow yields task placements.

allows the scheduler to jointly consider their placement, and thus to find the best trade-off for the whole batch. A natural extension of this idea is to reconsider the *entire* existing workload ("rescheduling"), and to preempt and migrate running tasks if prudent.

Flow-based scheduling, introduced by Quincy [22], is an efficient batching technique. Flow-based scheduling uses a placement mechanism – min-cost max-flow (MCMF) optimization – with an attractive property: it guarantees overall optimal task placements for a given scheduling policy. Figure 2b illustrates how it proceeds. If a change to cluster state happens (e.g., task submission), the scheduler updates an internal graph representation of the scheduling problem. It waits for any running optimization to finish, and then runs a MCMF solver on the graph. This yields an optimal flow from which the scheduler extracts the task assignments.

However, Figure 3 illustrates that Quincy, the current state-of-the-art flow-based scheduler, is too slow to meet our placement latency goal at scale. In this experiment, we replayed subsets of the public Google trace [30], which we augmented with locality preferences for batch processing jobs[2] against our faithful reimplementation of Quincy's approach. We measured the scheduler algorithm runtime for clusters of increasing size with proportional workload growth. The algorithm runtime increases with scale, up to a median of 64s and a 99th percentile of 83s for the full Google cluster (12,500 machines). During this time, the scheduler must wait for the solver to finish, and cannot choose any placements for new tasks.

The goal of this paper is to build a flow-based scheduler that achieves equal placement quality to Quincy, but which does so at sub-second placement latency. As our experiment illustrates, we must achieve at least an order-of-magnitude speedup over Quincy to meet this goal.

3 Firmament approach

We chose to develop Firmament as a flow-based scheduler for three reasons. First, flow-based scheduling considers the entire workload, allowing us to support rescheduling and priority preemption. Second, flow-based scheduling achieves high placement quality and, consequently, low task response times [22, §6]. Third, as a batching approach, flow-based scheduling amortizes work well over many tasks and placement decisions, and hence achieves high task throughput – albeit at a high placement latency that we aim to improve.

[2]Details of our simulation are in §7; in the steady-state, the 12,500-machine cluster runs about 150,000 tasks comprising about 1,800 jobs.

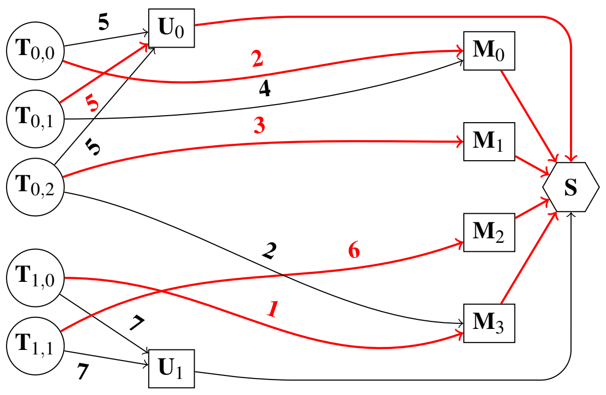


Figure 5: Example flow network for a four-machine cluster with two jobs of three and two tasks. All tasks except $\mathbf{T}_{0,1}$ are scheduled on machines. Arc labels show non-zero costs, and all arcs have unit capacity apart from those between unscheduled aggregators and the sink. The red arcs carry flow and form the min-cost solution.

3.1 Architecture

Figure 4 gives an overview of the Firmament scheduler architecture. Firmament, like Quincy, models the scheduling problem as a min-cost max-flow (MCMF) optimization over a *flow network*. The flow network is a directed graph whose structure is defined by the *scheduling policy*. In response to events and monitoring information, the flow network is modified according to the scheduling policy, and submitted to an *MCMF solver* to find an optimal (i.e., min-cost) flow. Once the solver completes, it returns the optimal flow, from which Firmament extracts the implied task placements. In the following, we first explain the basic structure of the flow network, and then discuss how to make the solver fast.

3.2 Flow network structure

A flow network is a directed graph whose arcs carry *flow* from source nodes to a sink node. A *cost* and *capacity* associated with each arc constrain the flow, and specify preferential routes for it.

Figure 5 shows an example of a flow network that expresses a simple cluster scheduling problem. Each task node $\mathbf{T}_{j,i}$ on the left hand side, representing the i^{th} task of job j, is a source of one unit of flow. All such flow must be drained into the sink node (**S**) for a feasible solution to the optimization problem. To reach **S**, flow from $\mathbf{T}_{j,i}$ can proceed through a machine node ($\mathbf{M}_m$), which schedules the task on machine m (e.g., $\mathbf{T}_{0,2}$ on $\mathbf{M}_1$). Alternatively, the flow may proceed to the sink through an unscheduled aggregator node ($\mathbf{U}_j$ for job j), which leaves the task unscheduled (as with $\mathbf{T}_{0,1}$) or preempts it if running.

In the example, a task's placement preferences are expressed as costs on direct arcs to machines. The cost to leave the task unscheduled, or to preempt it when running, is the cost on its arc to the unscheduled aggregator (e.g., 7 for $\mathbf{T}_{1,1}$). Given this flow network, an MCMF solver finds a globally optimal (i.e., minimum-cost) flow (shown in red in Figure 5). This optimal flow expresses the best trade-off between the tasks' unscheduled costs and their placement preferences. Task placements are extracted by tracing flow from the machines back to tasks.

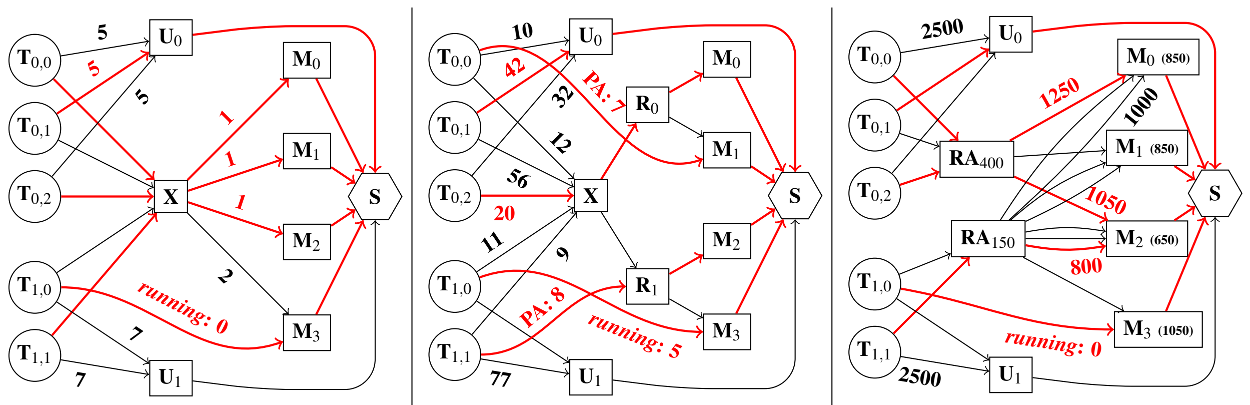
In our example, tasks had only direct arcs to machines. The solver finds the best solution if every task has an arc to each machine scored according to the scheduling policy, but this requires thousands of arcs per task on a large cluster. Policy-defined *aggregator* nodes, similar to the unscheduled aggregators, reduce the number of arcs required to express a scheduling policy. Such aggregators group, e.g., machines in a rack, tasks with similar resource needs, or machines with similar capabilities. With aggregators, the cost of a task placement is the sum of all costs on the path from the task node to the sink.

3.3 Scheduling policies

Firmament generalizes flow-based scheduling over the single, batch-oriented policy proposed by Quincy. Cluster administrators use a policy API to configure Firmament's scheduling policy, which may incorporate e.g., multi-dimensional resources, fairness, and priority preemption [31, Ch. 6–7]. This paper focuses on Firmament's scalability, and we therefore use only three simplified, illustrative policies explained in the following: (*i*) a simple load-spreading policy, (*ii*) Quincy's slot-based, locality-oriented policy, and (*iii*) a network-aware policy that avoids overloading machines' network connections.

Load-spreading policy. Figure 6a shows a trivial use of an aggregator: all tasks have arcs to a cluster-wide aggregator (**X**). The cost on the outgoing arc from **X** to each machine node is proportional to the number of tasks already running on the machine (e.g., one task on $\mathbf{M}_3$). The effect is that the number of tasks on a machine only increases once all other machines have at least as many tasks (as e.g., in Docker SwarmKit). This policy neither requires or nor uses the full sophistication of flow-based scheduling. We use it to highlight specific edge cases in MCMF algorithms (see §4.3).

Quincy policy. Figure 6b depicts Quincy's original locality-oriented policy [22, §4.2], which uses rack aggregators ($\mathbf{R}_r$) and a cluster aggregator (**X**) to express data locality for batch jobs. Tasks have low-cost *preference arcs* to machines and racks on which they have local data, but fall back to scheduling via the cluster aggregator if their preferences are unavailable (e.g., $\mathbf{T}_{0,2}$). This policy is suitable for batch jobs, and optimizes for a trade-off between data locality, task wait time, and preemption cost. We use it to illustrate MCMF algorithm performance and for head-to-head comparison with Quincy.



(a) Load-spreading policy with a single cluster aggregator (**X**) and costs proprtional to number of tasks per machine.

(b) Quincy policy with cluster (**X**) and rack (**R**) aggregators, and data locality preference arcs (PA).

(c) Network-aware policy with request aggregators (**RA**) and dynamic arcs to machines with spare network bandwidth.

Figure 6: Different aggregators, arcs, and costs help Firmament express the scheduling policies used in this paper; costs are example values consistent with each policy. Firmament also supports other policies via an API [31, Ch. 6–7].

Network-aware policy. Figure 6c illustrates a policy which avoids overcommitting machines' network bandwidth (which degrades task response time). Each task connects to a request aggregator (**RA**) for its network bandwidth request. The **RA**s have one arc for each task that fits on each machine with sufficient spare bandwidth (e.g., 650 MB/s of 1.25 GB/s on $\mathbf{M}_2$'s 10G link). These arcs are dynamically adapted as the observed bandwidth use changes. Costs on the arcs to machines are the sum of the request and the currently used bandwidth, which incentivizes balanced utilization. We use this policy to illustrate Firmament's potential to make high-quality decisions, but a production policy would be more complex and extend it with a priority notion and additional resource dimensions (e.g., CPU/RAM) [31, §7.3].

4 Min-cost max-flow algorithms

A flow-based scheduler can use any MCMF algorithm, but some algorithms are better suited to the scheduling problem than others. In this section, we explain the MCMF algorithms that we implemented for Firmament, compare them empirically, and explain their sometimes unexpected performance.

A min-cost max-flow algorithm takes a directed flow network $G = (N, A)$ as input. Each arc $(i, j) \in A$ has a cost c_{ij} and a maximum capacity u_{ij}. Each node $i \in N$ also has an associated supply $b(i)$; nodes with positive supply are *sources*, those with negative supply are *sinks*.

Informally, MCMF algorithms must optimally route the flow from all sources (e.g., task nodes $\mathbf{T}_{i,j}$) to sinks (e.g., the sink node **S**) without exceeding the capacity constraint on any arc. To understand the differences between MCMF algorithms, we need a slightly more formal definition: the goal is to find a flow f that minimizes Eq. 1, while respecting the flow *feasibility constraints* of **mass balance** (Eq. 2) and **capacity** (Eq. 3):

$$\text{Minimize} \sum_{(i,j)\in A} c_{ij} f_{ij} \text{ subject to} \tag{1}$$

$$\sum_{k:(j,k)\in A} f_{jk} - \sum_{i:(i,j)\in A} f_{ij} = b(j), \forall j \in N \tag{2}$$

$$\text{and } 0 \leq f_{ij} \leq u_{ij}, \forall (i,j) \in A \tag{3}$$

Some algorithms use an equivalent definition of the flow network, the *residual network*. In the residual network, each arc $(i, j) \in A$ with cost c_{ij} and maximum capacity u_{ij} is replaced by two arcs: (i, j) and (j, i). Arc (i, j) has cost c_{ij} and a *residual capacity* of $r_{ij} = u_{ij} - f_{ij}$, while arc (j, i) has cost $-c_{ij}$ and a residual capacity $r_{ji} = f_{ij}$. The feasibility constraints also apply in the residual network.

The *primal* minimization problem (Eq. 1) also has an associated *dual* problem, which some algorithms solve more efficiently. In the dual min-cost max-flow problem, each node $i \in N$ has an associated dual variable $\pi(i)$ called the *potential*. The potentials are adjusted in different, algorithm-specific ways to meet optimality conditions. Moreover, each arc has a *reduced cost* with respect to the node potentials, defined as:

$$c^{\pi}_{ij} = c_{ij} - \pi(i) + \pi(j) \tag{4}$$

A feasible flow is optimal if and only if at least one of three *optimality conditions* is met:

1. **Negative cycle optimality**: no directed negative-cost cycles exist in the residual network.

2. **Reduced cost optimality**: there is a set of node potentials π such that there are no arcs in the residual network with negative reduced cost (c^{π}_{ij}).

Algorithm	**Worst-case complexity**
Relaxation	$O(M^3CU^2)$
Cycle canceling	$O(NM^2CU)$
Cost scaling	$O(N^2M\log(NC))$
Successive shortest path	$O(N^2U\log(N))$

Table 1: Worst-case time complexities for min-cost max-flow algorithms. N is the number of nodes, M the number of arcs, C the largest arc cost and U the largest arc capacity. In our problem, $M > N > C > U$.

3. **Complementary slackness optimality**: there is a set of node potentials π such that the flow on arcs with $c^{\pi}_{ij} > 0$ is zero, and there are no arcs with both $c^{\pi}_{ij} < 0$ and available capacity.

Algorithms. The simplest MCMF algorithm is **cycle canceling** [25]. The algorithm first computes a max-flow solution, and then performs a series of iterations in which it augments flow along negative-cost directed cycles in the residual network. Pushing flow along such a cycle guarantees that the overall solution cost decreases. The algorithm finishes with an optimal solution once no negative-cost cycles remain (i.e., the negative cycle optimality condition is met). Cycle canceling always maintains feasibility and attempts to achieve optimality.

Unlike cycle canceling, the **successive shortest path** algorithm [2, p. 320] maintains reduced cost optimality at every step and tries to achieve feasibility. It repeatedly selects a source node (i.e., $b(i) > 0$) and sends flow from it to the sink along the shortest path.

The **relaxation** algorithm [4; 5], like successive shortest path, augments flow from source nodes along the shortest path to the sink. However, unlike successive shortest path, relaxation optimizes the dual problem by applying one of two changes when possible:

1. Keeping π unchanged, the algorithm modifies the flow, f, to f' such that f' still respects the reduced cost optimality condition and the total supply decreases (i.e., feasibility improves).
2. It modifies π to π' and f to f' such that f' is still a reduced cost-optimal solution and the cost of that solution decreases (i.e., total cost decreases).

This allows relaxation to decouple the improvements in feasibility from reductions in total cost. When relaxation can reduce cost or improve feasibility, it reduces cost.

Cost scaling [17–19] iterates to reduce cost while maintaining feasibility, and uses a relaxed complementary slackness condition called ε-optimality. A flow is ε-optimal if the flow on arcs with $c^{\pi}_{ij} > \varepsilon$ is zero and there are no arcs with $c^{\pi}_{ij} < -\varepsilon$ on which flow can be sent. Initially, ε is equal to the maximum arc cost, but ε rapidly decreases as it is divided by a constant factor after every iteration that achieves ε-optimality. Cost scaling finishes when $\frac{1}{n}$-optimality is achieved, since this is equivalent to the complementary slackness optimality condition [17].

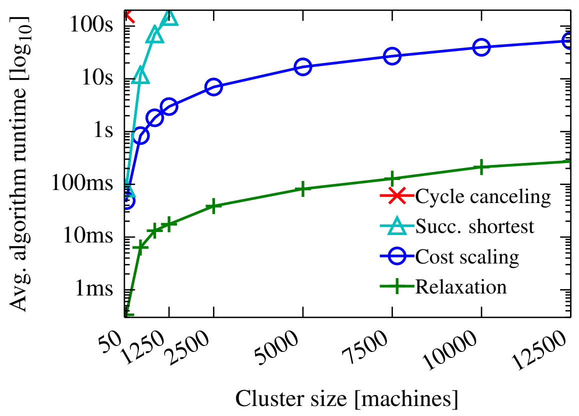


Figure 7: Average runtime for MCMF algorithms on clusters of different sizes, subsampled from the Google trace. We use the Quincy policy and slot utilization is about 50%. Relaxation performs best, despite having the highest time complexity. [*N.B.:* $\log_{10}$-scale y-axis.]

4.1 Algorithmic performance

Table 1 summarizes the worst-case complexities of the algorithms discussed. The complexities suggest that successive shortest path ought to work best, as long as $U\log(N) < M\log(NC)$, which is the case as $U \ll M$ and $C \geq 1$. However, since MCMF algorithms are known to have variable runtimes depending on the input graph [15; 24; 26], we decided to directly measure performance.

4.2 Measured performance

As in the experiment in Figure 3, we subsample the Google trace and replay it for simulated clusters of different sizes. We use the Quincy scheduling policy for batch jobs and prioritize service jobs over batch ones. Figure 7 shows the average runtime for each MCMF algorithm considered. Even though it has the best worst-case time complexity, successive shortest path outperforms only cycle canceling, and even on a modest cluster of 1,250 machines its algorithm runtime exceeds 100 seconds.

Moreover, the relaxation algorithm, which has the highest worst-case time complexity, actually performs best in practice. It outperforms cost scaling (used in Quincy) by two orders of magnitude: on average, relaxation completes in under 200ms even on a cluster of 12,500 machines. One key reason for this perhaps surprising performance is that relaxation does minimal work when most scheduling choices are straightforward. This happens if the destinations for tasks' flow are uncontested, i.e., few new tasks have arcs to the same location and attempt to schedule there. In this situation, relaxation routes most of the flow in a single pass over the graph.

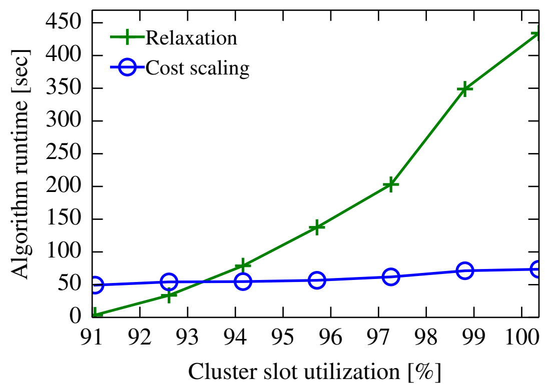


Figure 8: Close to full cluster utilization, relaxation runtime increases dramatically, while cost scaling is unaffected: the *x*-axis shows the utilization after scheduling jobs of increasing size to a 90%-utilized cluster.

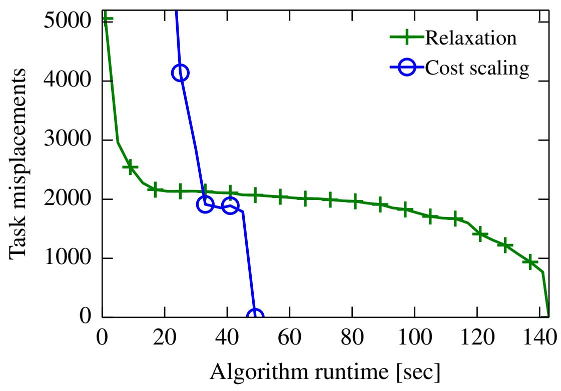


Figure 10: Approximate min-cost max-flow yields poor solutions, since many tasks are misplaced until shortly before the algorithms reach the optimal solution.

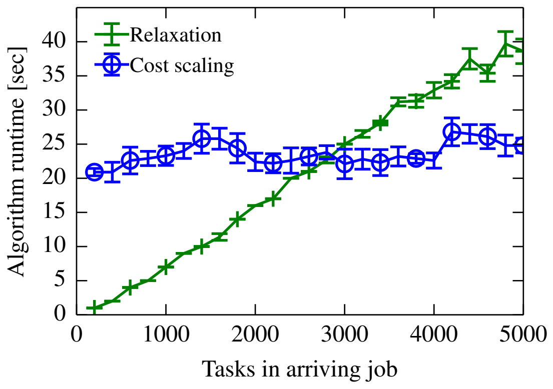


Figure 9: Contention slows down the relaxation algorithm: on cluster with a load-spreading scheduling policy, relaxation runtime exceeds that of cost scaling at just under 3,000 concurrently arriving tasks (e.g., a large job).

4.3 Edge cases for relaxation

Yet, relaxation is not always the right choice. For example, it can perform poorly under the high load and oversubscription common in batch analytics clusters [29]. Figure 8 illustrates this: here, we push the simulated Google cluster closer to oversubscription. We take a snapshot of the cluster and then submit increasingly larger jobs. The relaxation runtime increases rapidly, and at about 93% cluster utilization, it exceeds that of cost scaling, growing to over 400s in the oversubscribed case.

Moreover, some scheduling policies *inherently* create contention between tasks. Consider, for example, our load-spreading policy that balances the task count on each machine. This policy makes "under-populated" machines a popular destination for tasks' flow, and thus creates contention. We illustrate this with an experiment: we submit a single job with an increasing number of tasks to a cluster using the load-spreading policy. This corresponds to the rare-but-important arrival of very large jobs: for example, 1.2% of jobs in the Google trace have over 1,000 tasks, and some even over 20,000. Figure 9 shows that relaxation's runtime increases linearly in the number of tasks, and that it exceeds the runtime of cost scaling once the new job has over 3,000 tasks.

To make matters worse, a single overlong relaxation run can have a devastating effect on long-term placement latency. If many new tasks arrive during such a long run, the scheduler might again be faced with many unscheduled tasks when it next runs. Hence, relaxation may take a long time again, accumulate many changes, and in the worst case fail to ever recover to low placement latency.

5 MCMF optimizations for scheduling

Relaxation has promising common-case performance at scale for typical workloads. However, its edge-case behavior makes it necessary either (*i*) to fall back to other algorithms in these cases, or (*ii*) to reduce runtime in other ways. In the following, we use challenging graphs to investigate optimizations that either improve relaxation or the best "fallback" algorithm, cost scaling.

5.1 Approximate min-cost max-flow

MCMF algorithms return an *optimal* solution. For the cluster scheduling problem, however, an approximate solution may well suffice. For example, TetriSched [33] (based on an MILP solver), as well as Paragon [11] and Quasar [12] (based on collaborative filtering), terminate their solution search after a set time. We therefore investigated the solution quality of cost scaling and relaxation when they are terminated early. This would work well if the algorithms spent a long time on minor solution refinements with little impact on the overall outcome.

Algorithm	*Feasiblity*	*Red. cost optimality*	*ε-optimality*
Relaxation	–	✓	–
Cycle canceling	✓	–	–
Cost scaling	✓	–	✓
Succ. shortest path	–	✓	–

Table 2: Algorithms have different preconditions for each internal iteration. Cost scaling expects feasibility and ε-optimality, making it difficult to incrementalize.

In our experiment, we use a highly-utilized cluster (cf. Figure 8) to investigate relaxation and cost scaling, but the results generalize. Figure 10 shows the number of "misplaced" tasks as a function of how early we terminate the algorithms. We treat any task as misplaced if it is (*i*) preempted in the approximate solution but keeps running in the optimal one; (*ii*) scheduled on a different machine to where it is scheduled in the optimal solution. Both cost scaling and relaxation misplace thousands of tasks when terminated early, and tasks are still misplaced even in the final iteration before completion. Hence, early termination appears not to be a viable placement latency optimization for flow-based schedulers.

5.2 Incremental min-cost max-flow

Since cluster state does not change dramatically between subsequent scheduling runs, the MCMF algorithm might be able to reuse its previous state. In this section, we describe what changes are required to make MCMF algorithms work incrementally, and provide some intuition for which algorithms are suitable for incremental use.

All cluster events (e.g., task submissions, machine failures) ultimately reduce to three different types of *graph change* in the flow network:

1. **Supply** changes at nodes when arcs or nodes which previously carried flow are removed (e.g., due to machine failure), or when nodes with supply are added to the graph (e.g., at task submission).
2. **Capacity** changes on arcs if machines fail or (re)join the cluster. Note that arc additions and removals can also be modeled as capacity changes from and to zero-capacity arcs.
3. **Cost** changes on an arc when the desirability of routing flow via that arc changes; when these happen exactly depends on the scheduling policy.

Changes to the supply of a node, an arc's capacity, or its cost can invalidate the feasibility and optimality of an existing flow. Some MCMF algorithms require the flow to be feasible at every step and improve ε-optimality, while others require optimality to always hold and improve feasibility (Table 2). A solution must be optimal *and* feasible because an infeasible solution fails to route all flow, which leaves tasks unscheduled or erroneously preempts them, while a non-optimal solution misplaces tasks.

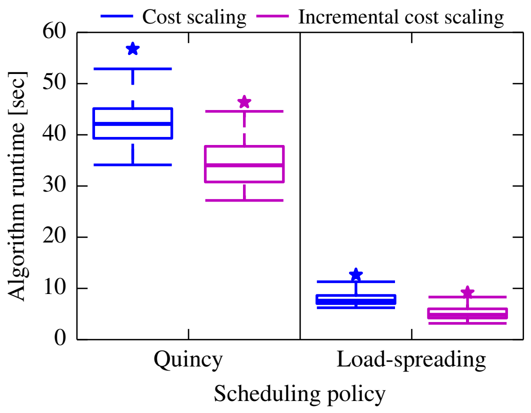


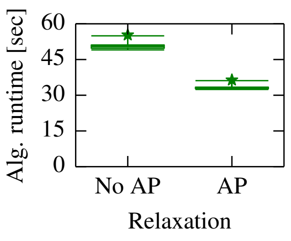
Figure 11: Incremental cost scaling is 25% faster compared to from-scratch cost scaling for the Quincy policy and 50% faster for the load-spreading policy.

	Reduced cost on arc from i to j		
Change type	$c^{\pi}_{ij} < 0$	$c^{\pi}_{ij} = 0$	$c^{\pi}_{ij} > 0$
Increasing arc cap.	🟥	🟩	🟩
Decreasing arc cap.	🟥	🟧 $f_{ij} > u\prime_{ij}$	🟩
Increasing arc cost	🟧 $c\prime^{\pi}_{ij} > 0$	🟧 $f_{ij} > 0$	🟩
Decreasing arc cost	🟩	🟥	🟧 $c\prime^{\pi}_{ij} < 0$

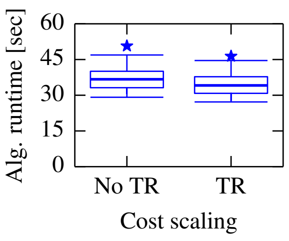
Table 3: Arc changes requiring solution reoptimization. *Green:* stays optimal and feasible; *red:* breaks feasibility or optimality; *orange:* breaks feasibility or optimality if condition in cell holds. Decreasing arc capacity can destroy feasibility; all other changes affect optimality only.

We implemented incremental versions of the cost scaling and relaxation algorithms. **Incremental cost scaling** is up to 50% faster than running cost scaling from scratch (Figure 11). Incremental cost scaling's potential gains are limited because cost scaling requires the flow to be feasible and ε-optimal before each intermediate iteration (Table 2). Graph changes can cause the flow to violate one or both requirements: for example, any addition or removal of task nodes adds supply and breaks feasibility. Table 3 shows the effect of different arc changes on the feasibility and optimality of the flow. A change that modifies the cost of an arc (i,j) from $c^{\pi}_{ij} < 0$ to $c\prime^{\pi}_{ij} > 0$, for example, breaks optimality. Many changes break optimality and cause cost scaling to fall back to a higher ε-optimality to compensate. To bring ε back down, cost scaling must do a substantial part of the work it would do from scratch. However, the limited improvement still helps reduce our fallback algorithm's runtime.

Incremental relaxation ought to work better than incremental cost scaling, since the relaxation algorithm only needs to maintain reduced cost optimality (Table 2). In practice, however, it turns out not to work well. While the algorithm can be incrementalized with relative ease and often runs faster, it – counter-intuitively – can also be



(a) Arc prioritization (AP).



(b) Eff. task removal (TR).

Figure 12: Problem-specific heuristics reduce runtime by 45% (AP, relaxation) and 10% (TR, inc. cost scaling).

slower incrementally than when running from scratch.

Relaxation requires reduced cost optimality to hold at every step of the algorithm and tries to achieve feasibility by pushing flow on zero-reduced cost arcs from source nodes to nodes with demand. The algorithm builds a tree of zero-reduced cost arcs from each source node in order to find such demand nodes. The tree is expanded by adding zero-reduced cost arcs to it. When running from scratch, the likelihood of zero-reduced cost arcs connecting two zero-reduced cost trees is low, as there are few such trees initially. Only when the solution is close to optimality, trees are joined into larger ones. Incremental relaxation, however, works with the existing, close-to-optimal state, which already contains large trees that must be extended for each source. Having to traverse these large trees many times, incremental relaxation can run slower than from scratch. This happens especially for graphs that relaxation already struggles with, e.g. ones that contain nodes with a lot of potential incoming flow. In practice, we found that incremental relaxation performs well only if tasks are not typically connected to a large zero-reduced cost tree.

5.3 Problem-specific heuristics

Our scheduler runs min-cost max-flow on a graph with specific properties, rather than the more general graphs typically used to evaluate MCMF algorithms [24, §4]. For example, our graph has a single sink; it is a directed acyclic graph; and flow must always traverse unscheduled aggregators or machine nodes. Hence, problem-specific heuristics might help the algorithms find solutions more quickly. We investigated several such heuristics, and found two beneficial ones: (*i*) prioritization of promising arcs, and (*ii*) efficient task node removal.

5.3.1 Arc prioritization

The relaxation algorithm builds a tree of zero-reduced cost arcs for every source node (see §5.2) in order to locate zero-reduced cost paths (i.e., paths that do not break reduced cost optimality) to nodes with demand. When this tree must be extended, any arc of zero reduced cost that connects a node inside the tree to a node outside the tree can be used. However, some arcs are better choices for extension than others. The quicker we can find paths to nodes with demand, the sooner we can route the supply. We therefore prioritize arcs that lead to nodes with demand when extending the cut, adding them to the front of a priority queue to ensure they are visited sooner.

In effect, this heuristic implements a hybrid graph traversal that biases towards depth-first exploration when demand nodes can be reached, but uses breadth first exploration otherwise. Figure 12a shows that applying this heuristic reduces relaxation runtime by 45% when running over a graph with contended nodes.

5.3.2 Efficient task removal

Our second heuristic helps incremental cost scaling. It is based on the insight that removal of a running task is common (e.g., due to completion, preemption, or a machine failure), but breaks feasibility. This happens because the task node is removed, which creates demand at the machine node where the task ran, since the machine node still has outgoing flow in the intermediate solution. Breaking feasibility is expensive for cost scaling (§5.2).

However, we can reconstruct the task's flow through the graph, remove it, and drain the machine node's flow at the single sink node. This creates demand in a single place only (the sink), which accelerates the incremental solution. However, Figure 12b shows that this heuristic offers only modest gains: it improves runtime by 10%.

6 Firmament implementation

We implemented a new MCMF solver for Firmament. It supports the four algorithms discussed earlier (§4) and incremental cost scaling. The solver consists of about 8,000 lines of C++. Firmament's cluster manager and our simulator are implemented in about 24,000 lines of C++, and are available at `http://firmament.io`.

In this section, we discuss implementation-level techniques that, in addition to our prior algorithmic insights, help Firmament achieve low task placement latency.

6.1 Algorithm choice

In §4, we saw that the practical performance of MCMF algorithms varies. Relaxation often works best, but scales poorly in specific edge cases. Cost scaling, by contrast, scales well and can be incrementalized (§5.2), but is usually substantially slower than relaxation.

Firmament's MCMF solver always speculatively executes cost scaling and relaxation, and picks the solution offered by whichever algorithm finishes first. In the common case, this is relaxation; having cost scaling as well guarantees that Firmament's placement latency does not grow unreasonably large in challenging situations. We run both algorithms instead of developing a heuristic to choose the right one for two reasons: first, it is cheap, as

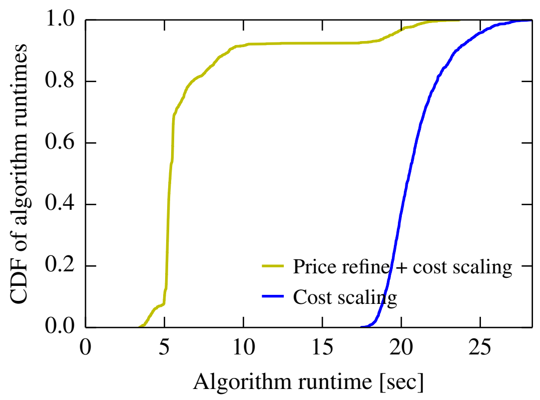


Figure 13: Incremental cost scaling runs 4× faster if we apply the price refine heuristic to a graph from relaxation.

the algorithms are single-threaded and do not parallelize; second, predicting the right algorithm is hard and the heuristic would depend on both scheduling policy and cluster utilization (cf. §4), making it brittle and complex.

6.2 Efficient algorithm switching

Firmament also applies an optimization that helps it efficiently transition state from relaxation to incremental cost scaling. Firmament's MCMF solver uses incremental cost scaling as it is faster than running cost scaling from scratch (§5.2). Typically, however, the (from-scratch) relaxation algorithm finishes first. The next incremental cost scaling run must therefore use the solution from a prior *relaxation* as a starting point. Since relaxation and cost scaling use different reduced cost graph representations, this can be slow. Specifically, relaxation may converge on node potentials that fit poorly into cost scaling's complementary slackness requirement, since relaxation only requires reduced cost optimality.

We found that *price refine* [17], a heuristic originally developed for use within cost scaling, helps with this transition. Price refine reduces the node potentials without affecting solution optimality, and thus simplifies the problem for cost scaling. Figure 13 shows that applying price refine to the prior relaxation solution graph speeds up incremental cost scaling by 4× in 90% of cases.

We apply price refine on the previous solution *before* we apply the latest cluster changes. This guarantees that price refine is able to find node potentials that satisfy complementary slackness optimality without modifying the flow. Consequently, cost scaling must start only at a value of ε equal to the costliest arc graph change.

6.3 Efficient solver interaction

So far, we have primarily focused on reducing the MCMF solver's algorithm runtime. To achieve low task placement latency, we must make two steps that fall outside the solver runtime efficient as well. First, Firmament must efficiently update the flow network's nodes, arcs, costs, and capacities before every MCMF optimization to reflect the chosen scheduling policy. Second, Firmament must quickly extract task placements out of the flow network after the optimization finishes. We improve over the prior work on flow-based scheduling in Quincy for both aspects, as explained in the following.

```
to_visit = machine_nodes # list of machine nodes
node_flow_destinations = {} # auxiliary remember set
mappings = {} # final task mappings
while not to_visit.empty():
  node = to_visit.pop()
  if node.type is not TASK_NODE:
    # Visit the incoming arcs
    for arc in node.incoming_arcs():
      moved_machines = 0
      # Move as many machines to the incoming arc's
      # source node as there is flow on the arc
      while assigned_machines < arc.flow:
        node_flow_destinations[arc.source].append(
          node_flow_destinations[node].pop())
        moved_machines += 1
      # (Re)visit the incoming arc's source node
      if arc.source not in to_visit:
        to_visit.append(arc.source)
  else: # node.type is TASK_NODE
    mappings[node.task_id] =
      node_flow_destinations[node].pop()
return mappings
```

Listing 1: Our efficient algorithm for extracting task placements from the optimal flow returned by the solver.

Flow network updates. Firmament does two breadth-first traversals of the flow network to update it for a new solver run. The first traversal updates resource statistics associated with every entity, such as the memory available on a machine, its current load, or a task's resource request. The traversal starts from the nodes adjacent to the sink (usually machine nodes), and propagates statistics along each node's incoming arcs. Upon the first traversal's completion, Firmament runs a second traversal that starts at the task nodes. This pass allows the scheduling policy to update the flow network's nodes, arcs, costs and capacities using the statistics gathered in the first traversal. Hence, only two passes over the large graph must be made to prepare the next solver run. Their overhead is negligible compared to the solver runtime.

Task placement extraction. At the end of a run, the solver returns an optimal flow through the given network and Firmament must extract the task placements implied by this flow. Since Firmament allows arbitrary aggregators in the flow network, paths from tasks to machines may be longer than in Quincy, where arcs necessarily pointed to machines or racks. Hence, we had to generalize Quincy's approach to this extraction [22, p. 275]. To extract task assignments efficiently, we devised the graph traversal algorithm shown in Listing 1. The algorithm starts from machine nodes and propagates a list of machines to which each node has sent flow via its incoming arcs. In the common case, the algorithm extracts the task placements in a single pass over the graph.

7 Evaluation

We now evaluate how well Firmament meets its goals:

1. How do Firmament's task placement quality and placement latency compare to Quincy's? (§7.2)
2. How does Firmament cope with demanding situations such as an overloaded cluster? (§7.3)
3. At what operating points does Firmament fail to achieve sub-second placement latency? (§7.4)
4. How does Firmament's placement quality compare to other cluster schedulers on a physical cluster running a mixed batch/service workload? (§7.5)

7.1 Methodology

Our experiments combine scale-up simulations with experiments on a local testbed cluster.

In **simulations**, we replay a public production workload trace from 12,500-machine Google cluster [30] against Firmament's implementation. Our simulator is similar to Borg's "Fauxmaster" [35, §3.1]: it runs Firmament's real code and scheduling logic against simulated machines, merely stubbing out RPCs and task execution. However, there are three important limitations to note. First, the Google trace contains multi-dimensional resource requests for each task. Firmament supports multi-dimensional feasibility checking (as in Borg [35, §3.2]), but in order to fairly compare to Quincy, we use slot-based assignment. Second, we do not enforce task constraints for the same reason, even though they typically help Firmament's MCMF solver. Third, the Google trace lacks information about job types and input sizes. We use Omega's priority-based job type classification [32, §2.1], and estimate batch task input sizes as a function of the known runtime using typical industry distributions [8].

In **local cluster experiments**, we use a homogeneous 40-machine cluster. Each machine has a Xeon E5-2430Lv2 CPU (12× 2.40GHz), 64 GB RAM, and uses a 1TB magnetic disk for storage. The machines are connected via 10 Gbps, full-bisection bandwidth Ethernet.

When we compare with **Quincy**, we run Firmament with Quincy's scheduling policy and restrict the solver to use only cost scaling (as Quincy's `cs2` solver does).

7.2 Scalability *vs.* Quincy

In Figure 3, we illustrated that Quincy fails to scale to clusters of thousands of machines at an acceptable placement latency. We now repeat the same experiment using Firmament on the full-scale simulated Google cluster. However, we increase the cluster slot utilization from the earlier experiment's 50% to 90% to make the setup more challenging for Firmament, and also tune the cost scaling-based MCMF solver for its best performance.[3]

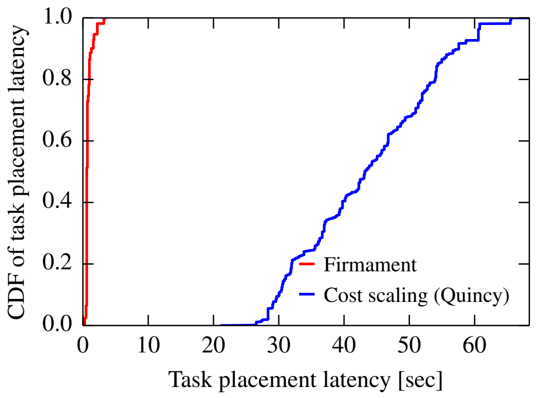


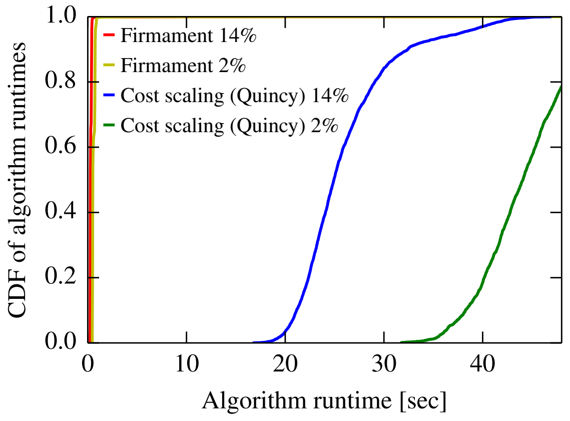
Figure 14: Firmament has a 20× lower task placement latency than Quincy on a simulated 12,500-machine cluster at 90% slot utilization, replaying the Google trace. The placement quality is identical to Quincy's.

Figure 14 shows the results as a CDF of *task placement latency*, i.e., the time between a task being submitted to the cluster manager and the time when it has been placed (§2). While Quincy takes between 25 and 60 seconds to place tasks, Firmament typically places tasks in hundreds of milliseconds and only exceeds a sub-second placement latency in the 90th percentile. Therefore, Firmament improves task placement latency by more than a 20× over Quincy, but maintains the same placement quality as it also finds an optimal flow.

Firmament's low placement latency comes because relaxation scales well even for large flow networks with the Google trace workload. This scalability allows us to afford scheduling policies with many arcs. As an illustrative example, we vary the data locality threshold in the Quincy scheduling policy. This threshold decides what fraction of a task's input data must reside on a machine or within a rack in order for the former to receive a preference arc to the latter. Quincy originally picked a threshold of a maximum of ten arcs per task. However, Figure 15a shows that even a lower threshold of 14% local data, which corresponds to at most seven preference arcs, yields algorithm runtimes of 20–40 seconds for Quincy's cost scaling. A low threshold allows the scheduler to exploit more fine-grained locality, but increases the number of arcs in the graph. Consequently, if we lower the threshold to 2% local data,[4] the cost scaling runtime in-

[3]Specifically, we found that an α-factor parameter value of 9, rather than the default of 2 used in Quincy, improves runtime by ≈30%.

[4]2% is a somewhat extreme value used for exposition here. The benefit of such a low threshold in a real cluster would likely be limited.



(a) Low preference thresholds see subs-second runtimes in Firmament, while Quincy (with cost scaling) takes over 40s.

Pref. threshold [local data]	*Input data locality*
14%	56%
2%	71%

(b) A lower preference threshold improves data locality.

Figure 15: Firmament scales to many arcs, and thus supports a lower preference arc threshold than Quincy.

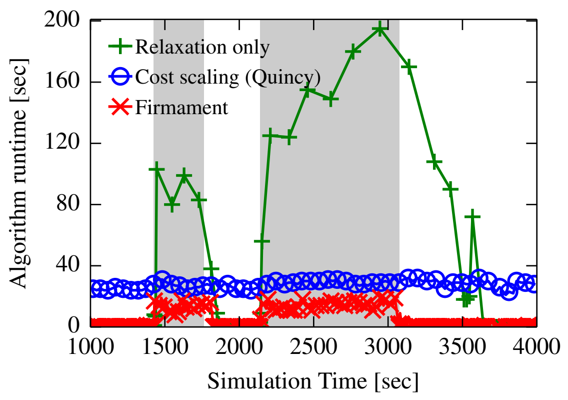


Figure 16: At times of high utilization (gray), Firmament outperforms relaxation and Quincy's cost scaling.

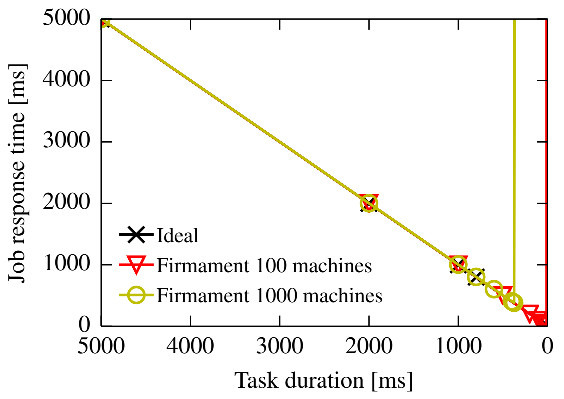


Figure 17: Firmament's breaking point is at tasks are shorter than ≈5ms at 100-machine scale, and ≈375ms at 1,000-machine scale, with 80% cluster slot utilization.

creases to well over 40 seconds. Firmament, on the other hand, still achieves sub-second algorithm even with a 2% threshold. This threshold yields an increase in data locality from 56% to 71% of total input data (Table 15b), which saves 4 TB of network traffic per simulated hour.

7.3 Coping with demanding situations

In the previous experiments, Firmament had a lower placement latency than Quincy because relaxation handles the Google workload well. As explained in §4, there are situations in which this is not the case. In those situations, Firmament picks incremental cost scaling's solution as it finishes first (§6). We now demonstrate the benefits of running two algorithms rather than just one.

In this experiment, we shrink the number of slots per cluster machine to reach 97% average utilization. Consequently, the cluster experiences transient periods of oversubscription. Figure 16 compares Firmament's automatic use of the fastest algorithm against using only one algorithm, either relaxation or cost scaling. During oversubscription, relaxation alone takes hundreds of seconds per run, while cost scaling alone completes in ≈30 seconds independent of cluster load. Firmament's incremental cost scaling finishes first in this situation, taking 10–15 seconds, which is about 2× faster than using cost scaling only (as Quincy does). Firmament also recovers earlier from the overload situation starting at 2,200s: while the relaxation-only runtime returns to sub-second level only around 3,700s, Firmament recovers at 3,200s. Relaxation on its own takes longer to recover because many tasks complete and free up slots during the long solver runs. These slots cannot be re-used until the next solver run completes, even though new, waiting tasks accumulate. Hence, Firmament's combination of algorithms outperforms either algorithm running alone.

7.4 Scalability to sub-second tasks

In the absence of oversubscription, we now investigate the scalability limit of Firmament's sub-second relaxation-based MCMF. To find Firmament's breaking point, we subject it to a worst-case workload consisting entirely of short tasks. This experiment is similar to Sparrow's breaking-point experiment for the centralized Spark scheduler [28, Fig. 12]. We submit jobs of 10 tasks at an interarrival time that keeps the cluster at a constant load of 80% if there is no scheduler overhead. We measure *job response time*, which is the maximum of the ten task response times for a job. In Figure 17, we plot job response time as a function of decreasing task duration. As

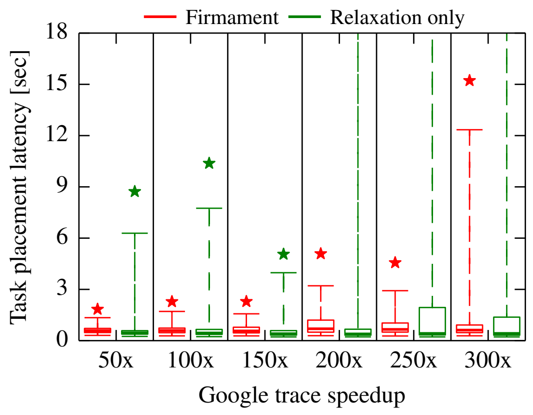


Figure 18: Firmament, unlike relaxation alone, keeps up with a 300× accelerated Google workload (1st, 25th, 50th, 75th, 99th percentiles and maximum).

we reduce task duration, we also reduce task interarrival time to keep the load constant, hence increasing the task throughput faced by the scheduler. With an ideal scheduler, job response time would be equal to task runtime as the scheduler would take no time to choose placements. Hence, the breaking point occurs when job response time deviates from the diagonal. For example, Spark's centralized task scheduler in 2013 had its breaking point on 100 machines at a 1.35 second task duration [28, §7.6].

By contrast, even though Firmament runs MCMF over the entire workload every time, Figure 17 shows that it achieves near-ideal job response time down to task durations as low as 5ms (100 machines) or 375ms (1,000 machines). This makes Firmament's response time competitive with distributed schedulers on medium-sized clusters that only run short tasks. At 10,000 machines, Firmament keeps up with task durations $\geq$5s. However, such large clusters usually run a mix of long-running and short tasks, rather than short tasks only [7; 10; 23; 35].

We therefore investigate Firmament's performance on a mixed workload. We speed up the Google trace by dividing all task runtimes and interarrival times by a speedup factor. This simulates a future workload of shorter batch tasks [27], while service jobs are still long-running. For example, at a 200× speedup, the median batch task takes 2.1 seconds, and the 90th and 99th percentile batch tasks take 18 and 92 seconds. We measure Firmament's placement latency across all tasks, and plot the distributions in Figure 18. Even at a speedup of 300×, Firmament keeps up and places 75% of the tasks at with sub-second latency. As before, a single MCMF algorithm does not scale: cost scaling's placement latency already exceeds 10s even without any speedup, and relaxation sees tail latencies well above 10 seconds beyond a 150× speedup, while Firmament scales further.

7.5 Placement quality on a local cluster

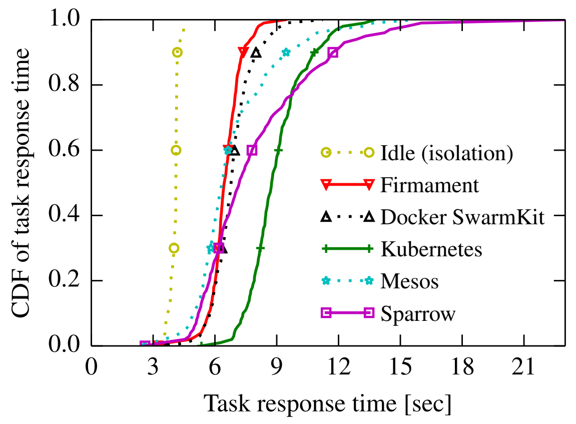
We deployed Firmament on a local 40-machine cluster to evaluate its real-world performance. We run a workload of short batch analytics tasks that take 3.5–5 seconds to complete on an otherwise idle cluster. Each task reads inputs of 4–8 GB from a cluster-wide HDFS installation in this experiment, and Firmament uses the network-aware scheduling policy. This policy reflects current network bandwidth reservations and observed actual bandwidth use in the flow network, and strives to place tasks on machines with lightly-loaded network connections. In Figure 19a, we show CDFs of task response times obtained using different cluster managers' schedulers. We measure task response time, and compare to a baseline that runs each task in isolation on an otherwise idle network. Firmament's task response time comes closest to the baseline above the 80th percentile as it successfully avoids overcommitting machines' network bandwidth. Other schedulers make random assignments (Sparrow), perform simple load-spreading (SwarmKit), or do not consider network bandwidth (Mesos, Kubernetes). Since our cluster is small, Firmament's task placement latency is inconsequential at around 5ms in this experiment.

Real-world clusters, however, run a mix of short, interactive tasks and long-running service and batch processing tasks. We therefore extend our workload with new long-running batch and service jobs to represent a similar mix. The long-running batch workloads are generated by fourteen `iperf` clients who communicate using UDP with seven `iperf` servers. Each `iperf` client generates 4 Gbps of sustained network traffic and simulates a batch job in a higher-priority network service class [20] than the short batch tasks (e.g., a TensorFlow [1] parameter server). Finally, we deploy three `nginx` web servers and seven HTTP clients as long-running service jobs. We run the cluster at about 80% network utilization, and again measure the task response time for the short batch analytics tasks. Figure 19b shows that Firmament's network-aware scheduling policy substantially improves the tail of the task response time distribution of short batch tasks. For example, Firmament's 99th percentile response time is 3.4× better than the SwarmKit and Kubernetes ones, and 6.2× better than Sparrow's. The tail matters, since the last task's response time often determines a batch job's overall response time (the "straggler" problem).

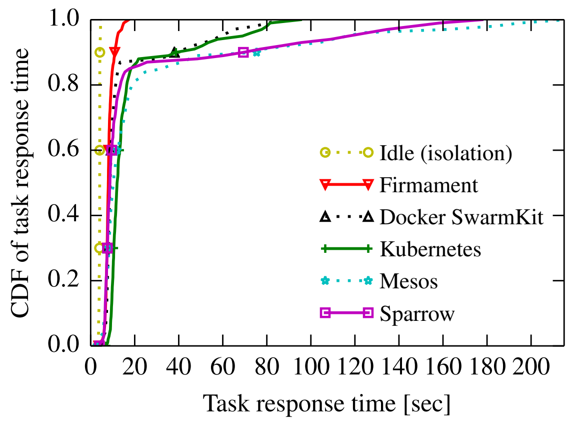
8 Related work

Many cluster schedulers exist, but Firmament is the first centralized one to offer high placement quality at sub-second placement latency on large clusters. We now briefly compare Firmament to existing schedulers.

Optimization-based schedulers. Firmament retains the same optimality as Quincy [22], but achieves much



(a) Short batch analytics tasks running on a cluster with an otherwise idle network. Overhead over "idle" due to contention.



(b) Short batch analytics tasks running on a cluster with background traffic from long-running batch and service tasks.

Figure 19: On a local 40-node cluster, Firmament improves task response time of short batch tasks in the tail using a network-aware scheduling policy, both (a) without and (b) with background traffic. Note the different *x*-axis scale.

lower placement latency. TetriSched [33] uses a mixed integer-linear programming (MILP) optimization and applies techniques similar to Firmament's (e.g., incremental restart from a prior solution) to reduce placement latency. Its placement quality degrades gracefully when terminated early (as required at scale), while Firmament always returns optimal solutions. Paragon [11], Quasar [12], and Bistro [16] also run expensive scoring computations (collaborative filtering, path selection), but scale the task placement by using greedy algorithms.

Centralized schedulers. Mesos [21] and Borg [35] match tasks to resources greedily; Borg's scoring uses random sampling with early termination [35, §3.4], which improves latency at the expense of placement quality. Omega [32] and Apollo [7] support multiple parallel schedulers to simplify their engineering and to improve scalability. Firmament shows that a single scheduler can attain scalability, but its MCMF optimization does not trivially admit multiple independent schedulers.

Distributed schedulers. Sparrow [28] and Tarcil [13] are distributed schedulers developed for clusters that see a high throughput of very short, sub-second tasks. In §7.4, we demonstrated that Firmament offers similarly low placement latency as Sparrow on clusters up to 1,000 machines, and beyond if only a part of the workload consists of short tasks. Mercury [23] is a hybrid scheduler that makes centralized, high-quality assignments for long tasks, and distributedly places short ones. With Firmament, we have shown that a centralized scheduler can scale even to short tasks, and that they benefit from the improved placement quality. Hawk [10] and Eagle [9] extend the hybrid approach with work-stealing and state gossiping techniques that improve placement quality; Yaq-d [29], by contrast, reorders tasks in worker-side queues to a similar end. Firmament shows that even a centralized scheduler can quickly schedule short tasks in large clusters with mixed workloads, rendering such complex compensation mechanisms largely unnecessary.

9 Conclusions

Firmament demonstrates that centralized cluster schedulers can scale to large clusters at low placement latencies. It chooses the same high-quality placements as an advanced centralized scheduler, at the speed and scale typically associated with distributed schedulers.

Firmament, our simulator, and our data sets are open-source and available from http://firmament.io. A Firmament scheduler plugin for Kubernetes [14] is currently under development.

Acknowledgements

We are grateful to M. Frans Kaashoek, Frank McSherry, Derek G. Murray, Rebecca Isaacs, Andrew Warfield, Robert Morris, and Pamela Delgado, as well as Jon Gjengset, Srivatsa Bhat, and the rest of MIT PDOS group for comments on drafts of this paper. We also thank Phil Gibbons, our shepherd, and the OSDI 2016 reviewers for their feedback. Their input much improved this paper.

This work was supported by a Google European Doctoral Fellowship, by NSF award CNS-1413920, and by the 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 paper 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.

References

[1] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, et al. "TensorFlow: A system for large-scale machine learning". In: *Proceedings of the 12th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*. Savannah, Georgia, USA, Nov. 2016.

[2] Ravindra K. Ahuja, Thomas L. Magnanti, and James B. Orlin. *Network flows: theory, algorithms, and applications*. Prentice Hall, 1993.

[3] Luiz André Barroso, Jimmy Clidaras, and Urs Hölzle. "The Datacenter as a Computer: An Introduction to the Design of Warehouse-Scale Machines, Second edition". In: *Synthesis Lectures on Computer Architecture* 8.3 (July 2013), pp. 1–154.

[4] Dimitri P. Bertsekas and Paul Tseng. "Relaxation Methods for Minimum Cost Ordinary and Generalized Network Flow Problems". In: *Operations Research* 36.1 (Feb. 1988), pp. 93–114.

[5] Dimitri P. Bertsekas and Paul Tseng. "The Relax codes for linear minimum cost network flow problems". In: *Annals of Operations Research* 13.1 (Dec. 1988), pp. 125–190.

[6] Arka A. Bhattacharya, David Culler, Eric Friedman, Ali Ghodsi, Scott Shenker, and Ion Stoica. "Hierarchical Scheduling for Diverse Datacenter Workloads". In: *Proceedings of the 4th Annual Symposium on Cloud Computing (SoCC)*. Santa Clara, California, Oct. 2013, 4:1–4:15.

[7] Eric Boutin, Jaliya Ekanayake, Wei Lin, Bing Shi, Jingren Zhou, Zhengping Qian, Ming Wu, et al. "Apollo: Scalable and Coordinated Scheduling for Cloud-Scale Computing". In: *Proceedings of the 11th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*. Broomfield, Colorado, USA, Oct. 2014, pp. 285–300.

[8] Yanpei Chen, Sara Alspaugh, and Randy Katz. "Interactive Analytical Processing in Big Data Systems: A Cross-industry Study of MapReduce Workloads". In: *Proceedings of the VLDB Endowment* 5.12 (Aug. 2012), pp. 1802–1813.

[9] Pamela Delgado, Diego Didona, Florin Dinu, and Willy Zwaenepoel. "Job-Aware Scheduling in Eagle: Divide and Stick to Your Probes". In: *Proceedings of the 7th ACM Symposium on Cloud Computing (SoCC)*. Santa Clara, California, USA, Oct. 2016.

[10] Pamela Delgado, Florin Dinu, Anne-Marie Kermarrec, and Willy Zwaenepoel. "Hawk: Hybrid Datacenter Scheduling". In: *Proceedings of the USENIX Annual Technical Conference*. Santa Clara, California, USA, July 2015, pp. 499–510.

[11] Christina Delimitrou and Christos Kozyrakis. "Paragon: QoS-aware Scheduling for Heterogeneous Datacenters". In: *Proceedings of the 18th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. Houston, Texas, USA, Mar. 2013, pp. 77–88.

[12] Christina Delimitrou and Christos Kozyrakis. "Quasar: Resource-Efficient and QoS-Aware Cluster Management". In: *Proceedings of the 18th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. Salt Lake City, Utah, USA, Mar. 2014.

[13] Christina Delimitrou, Daniel Sanchez, and Christos Kozyrakis. "Tarcil: Reconciling Scheduling Speed and Quality in Large Shared Clusters". In: *Proceedings of the 6th ACM Symposium on Cloud Computing (SoCC)*. Kohala Coast, Hawaii, USA, Aug. 2015, pp. 97–110.

[14] Cloud Native Computing Foundation. *Kubernetes*. `http://k8s.io`; accessed 14/11/2015.

[15] Antonio Frangioni and Antonio Manca. "A Computational Study of Cost Reoptimization for Min-Cost Flow Problems". In: *INFORMS Journal on Computing* 18.1 (2006), pp. 61–70.

[16] Andrey Goder, Alexey Spiridonov, and Yin Wang. "Bistro: Scheduling Data-Parallel Jobs Against Live Production Systems". In: *Proceedings of the USENIX Annual Technical Conference*. Santa Clara, California, USA, July 2015, pp. 459–471.

[17] Andrew V. Goldberg. "An Efficient Implementation of a Scaling Minimum-Cost Flow Algorithm". In: *Journal of Algorithms* 22.1 (1997), pp. 1–29.

[18] Andrew V. Goldberg and Michael Kharitonov. "On Implementing Scaling Push-Relabel Algorithms for the Minimum-Cost Flow Problem". In: *Network Flows and Matching: First DIMACS Implementation Challenge*. Ed. by D.S. Johnson and C.C. McGeoch. DIMACS series in discrete mathematics and theoretical computer science. American Mathematical Society, 1993.

[19] Andrew V. Goldberg and Robert E. Tarjan. "Finding Minimum-Cost Circulations by Successive Approximation". In: *Mathematics of Operations Research* 15.3 (Aug. 1990), pp. 430–466.

[20] Matthew P. Grosvenor, Malte Schwarzkopf, Ionel Gog, Robert N. M. Watson, Andrew W. Moore, Steven Hand, and Jon Crowcroft. "Queues don't matter when you can JUMP them!" In: *Proceedings of the 12th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*. Oakland, California, USA, May 2015.

[21] Benjamin Hindman, Andy Konwinski, Matei Zaharia, Ali Ghodsi, Anthony D. Joseph, Randy Katz, Scott Shenker, et al. "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)*. Boston, Massachusetts, USA, Mar. 2011, pp. 295–308.

[22] Michael Isard, Vijayan Prabhakaran, Jon Currey, Udi Wieder, Kunal Talwar, and Andrew Goldberg. "Quincy: fair scheduling for distributed computing clusters". In: *Proceedings of the 22nd ACM Symposium on Operating Systems Principles (SOSP)*. Big Sky, Montana, USA, Oct. 2009, pp. 261–276.

[23] Konstantinos Karanasos, Sriram Rao, Carlo Curino, Chris Douglas, Kishore Chaliparambil, Giovanni Matteo Fumarola, Solom Heddaya, et al. "Mercury: Hybrid Centralized and Distributed Scheduling in Large Shared Clusters". In: *Proceedings of the USENIX Annual Technical Conference*. Santa Clara, California, USA, July 2015, pp. 485–497.

[24] Zoltán Király and P. Kovács. "Efficient implementations of minimum-cost flow algorithms". In: *CoRR* abs/1207.6381 (2012).

[25] Morton Klein. "A Primal Method for Minimal Cost Flows with Applications to the Assignment and Transportation Problems". In: *Management Science* 14.3 (1967), pp. 205–220.

[26] Andreas Löbel. *Solving Large-Scale Real-World Minimum-Cost Flow Problems by a Network Simplex Method*. Tech. rep. SC-96-07. Zentrum für Informationstechnik Berlin (ZIB), Feb. 1996.

[27] Kay Ousterhout, Aurojit Panda, Joshua Rosen, Shivaram Venkataraman, Reynold Xin, Sylvia Ratnasamy, Scott Shenker, et al. "The case for tiny tasks in compute clusters". In: *Proceedings of the 14th USENIX Workshop on Hot Topics in Operating Systems (HotOS)*. Santa Ana Pueblo, New Mexico, USA, May 2013.

[28] Kay Ousterhout, Patrick Wendell, Matei Zaharia, and Ion Stoica. "Sparrow: Distributed, Low Latency Scheduling". In: *Proceedings of the 24th ACM Symposium on Operating Systems Principles (SOSP)*. Nemacolin Woodlands, Pennsylvania, USA, Nov. 2013, pp. 69–84.

[29] Jeff Rasley, Konstantinos Karanasos, Srikanth Kandula, Rodrigo Fonseca, Milan Vojnovic, and Sriram Rao. "Efficient Queue Management for Cluster Scheduling". In: *Proceedings of the 11th ACM European Conference on Computer Systems (EuroSys)*. London, United Kingdom, Apr. 2016.

[30] Charles Reiss, Alexey Tumanov, Gregory R. Ganger, Randy H. Katz, and Michael A. Kozuch. "Heterogeneity and dynamicity of clouds at scale: Google trace analysis". In: *Proceedings of the 3rd ACM Symposium on Cloud Computing (SoCC)*. San Jose, California, Oct. 2012, 7:1–7:13.

[31] Malte Schwarzkopf. "Operating system support for warehouse-scale computing". PhD thesis. University of Cambridge Computer Laboratory, Oct. 2015.

[32] Malte Schwarzkopf, Andy Konwinski, Michael Abd-El-Malek, and John Wilkes. "Omega: flexible, scalable schedulers for large compute clusters". In: *Proceedings of the 8th ACM European Conference on Computer Systems (EuroSys)*. Prague, Czech Republic, Apr. 2013, pp. 351–364.

[33] Alexey Tumanov, Timothy Zhu, Jun Woo Park, Michael A. Kozuch, Mor Harchol-Balter, and Gregory R. Ganger. "TetriSched: global rescheduling with adaptive plan-ahead in dynamic heterogeneous clusters". In: *Proceedings of the 11th ACM European Conference on Computer Systems (EuroSys)*. London, England, United Kingdom, 2016.

[34] Vinod Kumar Vavilapalli, Arun C. Murthy, Chris Douglas, Sharad Agarwal, Mahadev Konar, Robert Evans, Thomas Graves, et al. "Apache Hadoop YARN: Yet Another Resource Negotiator". In: *Proceedings of the 4th Annual Symposium on Cloud Computing (SoCC)*. Santa Clara, California, Oct. 2013, 5:1–5:16.

[35] Abhishek Verma, Luis David Pedrosa, Madhukar Korupolu, David Oppenheimer, and John Wilkes. "Large scale cluster management at Google". In: *Proceedings of the 10th ACM European Conference on Computer Systems (EuroSys)*. Bordeaux, France, Apr. 2015.

[36] Xiao Zhang, Eric Tune, Robert Hagmann, Rohit Jnagal, Vrigo Gokhale, and John Wilkes. “CPI^2: CPU Performance Isolation for Shared Compute Clusters”. In: *Proceedings of the 8th ACM European Conference on Computer Systems (EuroSys)*. Prague, Czech Republic, Apr. 2013, pp. 379–391.

Morpheus: Towards Automated SLOs for Enterprise Clusters

Sangeetha Abdu Jyothi [m,u] Carlo Curino[m] Ishai Menache[m]
Shravan Matthur Narayanamurthy[m] Alexey Tumanov[m,c] Jonathan Yaniv[t]
Ruslan Mavlyutov[m,f] Íñigo Goiri[m] Subru Krishnan[m] Janardhan Kulkarni[m]
Sriram Rao[m]

[m] Microsoft, [u] University of Illinois at Urbana–Champaign, [c] Carnegie Mellon University
[t] Technion-Israel Institute of Technology, [f] University of Fribourg

Abstract

Modern resource management frameworks for large-scale analytics leave unresolved the problematic tension between high cluster utilization and job's performance predictability—respectively coveted by operators and users. We address this in *Morpheus*, a new system that: 1) codifies implicit user expectations as explicit Service Level Objectives (SLOs), inferred from historical data, 2) enforces SLOs using novel scheduling techniques that isolate jobs from sharing-induced performance variability, and 3) mitigates inherent performance variance (e.g., due to failures) by means of dynamic reprovisioning of jobs. We validate these ideas against production traces from a 50k node cluster, and show that *Morpheus* can lower the number of deadline violations by 5× to 13×, while retaining cluster-utilization, and lowering cluster footprint by 14% to 28%. We demonstrate the scalability and practicality of our implementation by deploying *Morpheus* on a 2700-node cluster and running it against production-derived workloads.

1 Introduction

Commercial enterprises ranging from Fortune-500 companies to venture-capital funded startups are increasingly relying on multi-tenanted clusters for running their business-critical data analytics jobs. These jobs comprise of multiple tasks that are run on different cluster nodes, where the unit of per-task resource allocation is a container (i.e, a bundle of resources such as CPU, RAM and disk I/O) on an individual machine. From an analysis of large-scale production workloads, we observe significant variance in job runtimes, which sometimes results in missed deadlines and negative business impact. This is perceived by users as an *unpredictable* execution experience, and it accounts for 25% of (resource-provisioning related) user escalations in Microsoft big-data clusters.

Unpredictability comes from several sources, which for discussion purposes, we roughly group as follows:

- *Sharing-induced* – performance variability caused by inconsistent allocations of resources across job runs—a scheduling policy artifact.
- *Inherent* – performance variability due to changes in the job input (size, skew, availability), source code tweaks, failures, and hardware churn—this is endemic even in dedicated and lightly used clusters.

Unpredictability is most noticeable to users who submit *periodic* jobs (i.e., scheduled runs of the same job on newly arriving data). Their recurrent nature prompts users to form an expectation on jobs' runtime performance as well as react to any deviation from it, particularly, if the job is business-critical (i.e., a production job).

Unfortunately, widely deployed resource managers [9, 27, 51, 55] provide limited mechanisms (e.g., fairness weights, priorities, job killing) for users to cope with unpredictability of such jobs. Given these basic tools, users resort to a combination of ad-hoc tricks, often pivoting around conservative over-provisioning for important production jobs. These coarse compensating actions are manual and inherently error-prone. Worse, they may adversely impact cluster *utilization*—a key metric for cluster operators. Owing to the substantial costs involved in building/-operating large-scale clusters, operators seek good return on investment (ROI) by maximizing utilization.

Divergent predictability and utilization requirements are poorly handled by existing systems. This is taxing and leads to tension between users and operators.

An ideal resource management infrastructure would provide predictable execution as a core primitive, while achieving high cluster utilization. This is a worthwhile infrastructure to build, particularly, because periodic, production jobs make up the majority of cluster workloads, as reported by [43] and as we observe in §2.

In this paper, we move the state of the art towards this ideal, by proposing a system called *Morpheus*. Building *Morpheus* poses several interesting challenges such as, automatically: 1) capturing user predictability expectations, 2) controlling sharing-induced unpredictability, and 3) coping with inherent unpredictability. We elaborate on these challenges next.

Inferring SLOs and modeling job resource demands. Our first challenge is to formalize the *implicit* user predictability expectation in an *explicit* form that is actionable for the underlying resource management infrastructure. We refer to the resulting characterization as an (inferred) Service Level Objective (SLO). We focus on completion time SLOs or deadlines. The next step consists of quantifying the amount of resources that must be provisioned during the execution of the job to meet the SLO without wastefully over-provisioning resources. Naturally, the precise resource requirements of each job depend on numerous factors such as function being computed, the degree of parallelism, data size and skew.

The above is hard to accomplish for arbitrary jobs for two reasons: 1) target SLOs are generally unknown to operators, and often hard to define even for the users—see §2, and 2) automatic provisioning is a known hard problem even when fixing the application framework [52, 15, 26, 19]. However, the periodic nature of our workload makes this problem tractable by means of history-driven approaches. We tackle this problem using a combination of techniques: First, we statistically derive a target SLO for a periodic job by analyzing all inter-job data dependencies and ingress/egress operations (§ 4). Second, we leverage telemetry of historical runs to derive a job resource model—a time-varying skyline of resource demands. We employ a Linear Programming formulation, that explicitly controls the penalty of over/under provisioning—balancing predictability and utilization (§5). Programmatically deriving SLOs and job resource model enables a *tuning-free* user experience, where users can simply sign-off on the proposed contract. Users may alternatively override any parameter of the inferred SLO and the job resource model, which becomes binding if accepted by our system.

Eliminating sharing-induced unpredictability. Our second challenge is to enforce SLOs while retaining high-utilization in a shared environment. This consists of controlling performance variance with minimal resource over-provisioning. As noted above, sharing-induced unpredictability is a scheduling artifact. Accordingly, we structurally eliminate it by leveraging the notion of *recurring reservation*, a scheduling construct that isolates periodic production jobs from the noisiness of sharing. A key property of recurring reservations is that once a periodic job is admitted each of its instantiations will have a predictable resource allocation. High-utilization is achieved by means of a new online, planning algorithm (§ 6). The algorithm leverages jobs' flexibility (e.g., deadline slack) to pack reservations tightly.

Mitigating inherent unpredictability. Our last challenge is dealing with inherent performance variance (i.e., exogenous factors, such as task failures, code/data changes, etc.). We do this by dynamically re-provisioning the current instance of a reservation, in response to job resource consumption, in relationship to the SLO. This compensates for short-term drifts, while continuous retraining of our SLO and job resource model extractors captures long-term effects. This problem is in spirit similar to what was proposed in Jockey [19], as we discuss in §7.

We emphasize that all of the above techniques are framework-independent—this is key for our production clusters as they support multiple application frameworks.

Experimental validation. We validate our design by implementing *Morpheus* atop of Hadoop/YARN [51] (§8). We then perform several faithful simulations with traces of a production cluster with over 50k nodes, and show that the SLOs we derived are representative of the job's needs. The combination of tight job provisioning, reservation packing, and dynamic reprovisioning allows us to achieve: $5\times$ to $13\times$ reduction in potential SLO violations (with respect to user-defined static provisioning), and identical cluster utilization. All while, our packing algorithms leverage the flexibility in target SLOs to smooth the provisioning load over time, and achieve better ROI, by reducing cluster footprint by 14% to 28%. We conclude by deploying *Morpheus* on a 2700-node cluster, and performing stress-tests with a production-derived workload. This confirms both the scalability of our design, and the practicality of our implementation (§ 9). We intend to release components of *Morpheus* as open-source and the progress can be tracked at [2].

2 Motivation

In the early phases of our project, we set out to confirm/deny our informal intuitions of how big-data clusters are operated and used. We did so by analyzing four data sources: 1) execution logs of millions of jobs running on clusters with more than 50k nodes, 2) infrastructure deployment/upgrade logs, 3) interviews, discussion threads, and escalation tickets from users, operators and decision makers, and 4) targeted micro-benchmarks. We summarize below the main findings of our analysis.

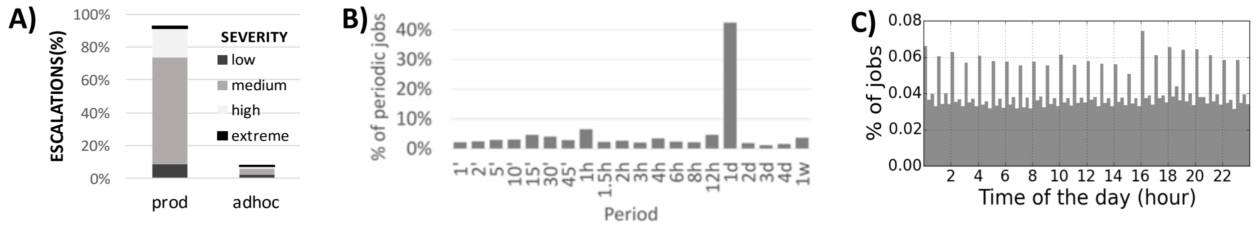


Figure 1: Analysis of user escalations and recurrent behaviors of production workloads.

2.1 Cluster workloads

Proper execution of production jobs is crucial. Production jobs represent over 75% of our workload and a similar percentage of the provisioned capacity—the rest being dedicated to ad-hoc jobs (10-20%) and ready to handle growth/failures (5-10%). All unassigned capacity is redistributed fairly to speed up jobs. As expected, users care mostly about proper execution of production jobs. Fig. 1a shows that over 90% of all escalations relate to production jobs, and this percentage grows to 100% for high/extreme severity escalations.

Predictability trumps fairness. Further analysis of the escalations of Fig. 1a and of discussion threads, indicates that users are 120$\times$ more likely to complain about the performance (un)predictability (25% of all job/resource-management escalations) than about fairness ($< 0.2\%$), despite the fact that our system does not enforce fairness strictly. This outcome may be expected, as customers cannot observe how "fair" allocations really are.

Production jobs are often periodic. Over 60% of the jobs in our larger/busier clusters are recurrent. Most of these recurring jobs are production jobs operating on continuously arriving data, hence are periodic in nature. Fig. 1b shows the distribution of the period for periodic jobs. Interestingly, most of the distribution mass is contributed by a small number of natural values (e.g., once-a-day, once-an-hour, etc.); this property will be useful to our allocation mechanisms (§6). Fig. 1c provides further evidence of recurrent behavior, by showing that job start times are more densely distributed around the "start-of-the-hour". This confirms that most jobs are submitted automatically on a fixed schedule.

The above evidence confirms that the most important portion of our workloads is strongly recurrent. This allows for planning the cluster agenda, without being overly conservative in the resource provisioning of jobs.

2.2 Predictability challenges

Manual tuning of job allocation is hard. Fig. 2a shows the distribution of the ratio between the total amount of resources provisioned by the job's owner and the job's actual resource usage (both comparing peak parallelism and area). The wide range of over/under-allocation indicates that it is very hard for users (or they lack incentives) to optimally provision resources for their jobs. We further validate this hunch through a user study in [15]. The graphs shows that 75% of jobs are over-provisioned (even at their peak), with 20% of them over 10$\times$ over-provisioned. This is likely due to users statically setting their provisioning for a periodic job. We confirm this, by observing that in one-month period over 80% of periodic jobs had no changes in their resource provisioning. Large under-provisioned jobs partially offset the impact of over-provisioning on cluster utilization.

Sources of performance variance. It is hard to precisely establish the sources of variance from the production logs we have. We observe a small but positive correlation (0.16) between the amount of sharing (above provisioned resources) and job runtime variance. This indicates that increased sharing affects runtime variance.

We investigate further the roles of sharing-induced and inherent performance variance by means of a simple micro-benchmark. Fig. 2b shows the normalized runtime of 5 TPC-H queries[1]. We consider two configurations one with constrained parallelism (500 containers), and one with unconstrained parallelism (>2000 containers); each container is a bundle of <1core,8GB RAM>. Each query was run 100 times in each configuration on an empty cluster at 10TB scale. The graph shows that even when removing common sources of inherent variability (data availability, failures, network congestion), runtimes remain unpredictable (e.g., due to stragglers, §7).

By analyzing these experiments and observing production environments, we conclude that: 1) history-based approaches can model well the "normal" behavior of a query (small box), 2) handling outliers (as in the long whiskers) without wasting resources requires a dynamic component that performs reprovisioning online, and 3) while each source of variance may be addressed with an ad-hoc solution, providing a general-purpose line of defense is paramount— see §7 for our solution.

[1]Box shows [25th,75th] percentiles, and whiskers shows [min,max].

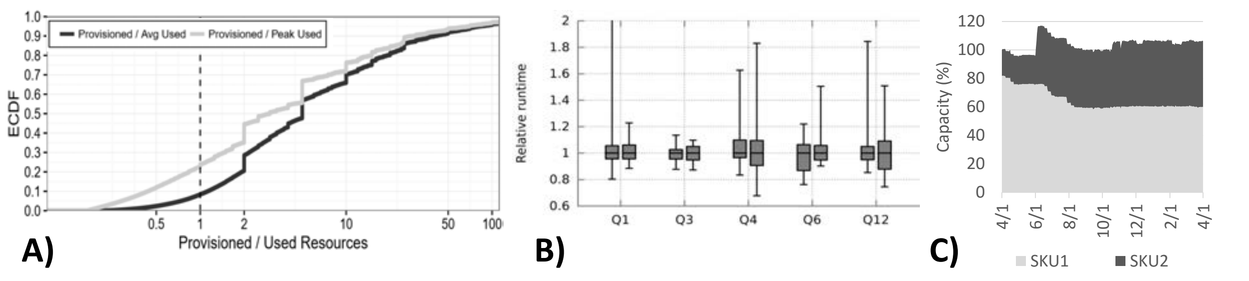


Figure 2: A) Empirical CDF of provisioning vs. used resources, B) box-whisker plot of normalized runtime of TPC-H queries running with 500 containers (left) and >2000 containers (right). C) cluster capacity of different machine types.

2.3 Changing conditions

Cluster conditions keep evolving—jobs may run on different server types. We provide in Fig. 2c a measure of hardware churn in our clusters. We refer to different machines configurations as Stock Keeping Units (SKUs). Over a period of a year, the ratio between number of machines of type SKU1 and type SKU2 changed from 80/20 to 55/45; the total number of nodes also kept changing over that period. This is notable, because even seemingly minor hardware differences can impact job runtime significantly— e.g., 40% difference in runtime on SKU1 vs SKU2 for a Spark production job.

User scripts keep evolving. We perform an analysis of the versioning of user scripts/UDFs. We remove all simple parameterizations that naturally change with every instantiation, and then construct a fuzzy match of the code structure. Within one-month of trace data, we detect that 15-20% of periodic jobs had at least one large code delta (more than 10% code difference), and over 50% had at least one small delta (any change that breaks MD5 of the parameter-stripped code). Hence, even an optimal static tuning is likely going to drift out of optimality over time.

Motivated by all of the above evidence, we focus on building a resource management substrate that provides predictable execution as a core primitive.

3 Overview of *Morpheus*

Morpheus is a system that continuously observes and learns as periodic jobs execute over time. The findings are used to economically reserve resources for the job ahead of job execution, and dynamically adapt to changing conditions at runtime. To give an informal sense of the key functionalities in *Morpheus*, we start our overview by following a typical life-cycle of a periodic job (JobX) as it is governed by *Morpheus* (§3.1). Next, we describe the core subsystems (§3.2). Fig. 3 provides a logical view of the architecture, and "zooms in" on a particular job.

3.1 "Life" of a periodic job

With reference to Fig. 3, a typical periodic jobs goes through the following stages.

1. The user periodically submits JobX with manually provisioned resources. In the meantime, the underlying infrastructure captures:
   (a) Data-dependencies and ingress/egress operations in the Provenance Graph (PG).
   (b) Resource utilization of each run (marked as the `R1-R4` skylines in Fig. 3) in a Telemetry-History (TH) database.
2. The SLO Inference performs an offline analysis of the successful runs of JobX:
   (a) From the PG it derives a deadline `d`—the *SLO*.
   (b) From the TH, it derives a model of the job resource demand over time, `R`*. We refer to `R`* as the *job resource model*
3. The user signs off (or optionally overrides) the automatically-generated SLO and job resource model.
4. *Morpheus* enforces SLOs via recurring reservations:
   (a) Adds a recurring reservation for JobX into the cluster agenda—this sets aside resources over time based on the job resource model `R`*.
   (b) New instances of JobX run within the recurring reservation (dedicated resources).
5. The Dynamic Reprovisioning component monitors the job progress online, and increases/decreases the reservation, to mitigate inherent execution variability.
6. Morpheus constantly feeds back into Step 2 the PG and telemetry information of the new runs for continuous learning and refinement of the SLO and the job resource model.

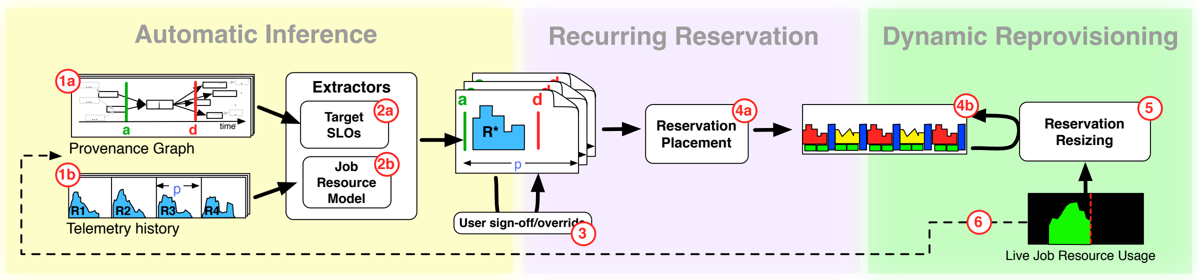


Figure 3: Conceptual view of *Morpheus*' architecture. Numbers/letters match the "Life" of a periodic job (§3.1).

3.2 Key components

We next give a brief overview of the different components in *Morpheus*, highlighting the different timescales in which they operate. Details follow in §4–§7. We start our description with the **automatic inference module**, which consists of two sub-components:

Extractor of target SLOs (§4). This sub-component operates on a Provenance Graph (PG). This is a graph capturing all inter-job data dependencies, as well as all ingress/egress operations. The SLO extractor leverages the precise timing information stored in the PG to statistically derive a deadline for the periodic job—as time at which downstream consumers read a job's output[2].

Job resource model (§5). This sub-component takes as input detailed telemetry information on job runs. The information includes time-series of resource utilization ("skyline")—the amount of resources (represented as containers) used by the job at certain time granularity, typically one minute. Based on time-series of multiple runs, the sub-component constructs a tight envelope $\mathtt{R}^*$ around the "typical" behavior of the job. These time-series are also used to derive the period, $\mathtt{P}$.

The automatic inference module outputs the SLO and the job resource model in the form of a *recurring reservation request* for a newly observed periodic jobs, and continuously refines existing ones on slow time scale (e.g., daily). The inferred SLO and job resource model feed the resource reservation component automatically, yet *Morpheus* also allows for **user SLO and job resource model sign-off/override**. More specifically, the job owners are given three options: 1) sign-off on the proposed SLO and job resource model as-is, 2) override any of the parameters based on further knowledge (e.g., the job will run on 10× more data starting tomorrow), or simply 3) reject the use of SLOs, in which case the job runs with standard fair-queueing semantics [51]. By signing off a recurring reservation the user approves the SLO, the initial job demand skyline, as well as it accepts a bounded (and configurable[3]) amount of runtime reprovisioning—more below.

Reservation Placement (§6). The SLO and the job resource model are expressed in terms of a recurring reservation request to a Reservation Placement mechanism. *Morpheus* implementation (§8) builds upon YARN's reservation system [51, 13], which we extend to accommodate periodic reservations. The allocation problem itself poses a substantial algorithmic challenge, as the goal is to "pack" efficiently online arriving jobs with different periods and arbitrary skylines. To address this challenge, we design a novel online packing algorithm specialized for periodic jobs. The algorithm exploits the jobs' flexibility (e.g., deadline slack) to compactly pack them, which leaves enough capacity for ad-hoc jobs. Our algorithm is incremental as it places new reservation requests, without modifying the allocation plan for other jobs.

Dynamic Reprovisioning (§7). Naturally, any tight and static capacity reservation cannot perfectly accommodate all job instances. To cope with dynamic variability in job execution (or "inherent unpredictability"), this component continuously monitors the rate of progress of the job with respect to the amount of reservation consumed. If progress appears slower/faster than expected, the component automatically adjusts the reservation, by tweaking the resources provisioned for this reservation. Notably, such black-box approach is framework-independent, which is key given the large amount of frameworks that run in our clusters.

3.3 Current limitations

Before we fully describe *Morpheus*, we briefly highlight some limitations of our system.

Control over globally-shared resources. *Morpheus* relies on the underlying resource management infrastructure

[2]Note that a small number of periodic jobs exhibit latency-sensitive behaviors (output consumed immediately). Our system handles those as a special case of a deadline with no slack.

[3]This is currently a system-wide parameter, but it could be easily evolved to be a per-job parameter if demanded by customers.

(Apache Hadoop/YARN in our current implementation) to enforce its decisions. As such, *Morpheus* can only enforce container-level resources (such as CPU/Memory), but lacks control over globally-shared resources (e.g., bandwidth on switches, DNS server). Resulting runtime variability is coped with via dynamic reprovisioning (§7).

Support of non-periodic jobs. *Morpheus* supports both periodic and non-periodic reservations, but does not automate the SLO and job resource model extraction for never-seen-before jobs. Recent literature has shown that job resource modeling can be performed a-priori (from query and input data only) for a given application framework [15, 42, 41, 56]—we discuss integration of these approaches in *Morpheus* in §8. SLO extraction for never-seen-before jobs remains an open problem.

Automatic SLAs. *Morpheus* provides an important building blocks towards, but does not aim at delivering full-fledged automated Service Level Agreements (SLAs). A full SLA typically includes a specification of the economic (or business) aspects of the provider-user agreement [57]. For example, it may include the cost of unit of resources, threshold of expected SLO attainment, legal/financial consequences of missing the target SLO, limits of how much dynamic-reprovisioning is allowed and charging consequences, etc. These aspects require further investigation beyond the scope of this paper.

4 SLO inference

In this section, we show how to automatically derive SLOs for periodic jobs based on inter-job dependencies.

Based on interviews with cluster operators and users, we isolate one observable metric which users care about: job *completion by a deadline*. Specifically, analyzing the escalation tickets, some users seem to form expectations such as: *"95% of job X runs should complete by 5pm"*. Other users are not able to specify a concrete deadline, but do state that other teams rely on the output of their job, and may need it in a timely manner. Overall, the goal of *Morpheus* is to crystallize what users perceive as "good-enough" job performance through automatically-generated target SLO. Towards that end, *Morpheus* utilizes a Provenance Graph (PG) as the main inference tool. We next briefly describe the inference procedure.

Provenance Graph – reasoning about cluster data. The PG gathers logs (petabytes daily across our production environments) capturing key aspects of job execution, file system accesses, and system metrics. The PG is a semantically rich and compact (few TBs) graph representation of these raw logs. Specifically, *nodes* represent jobs and files in our clusters, and *edges* capture read/write operations among jobs, files, and all ingress/egress operations

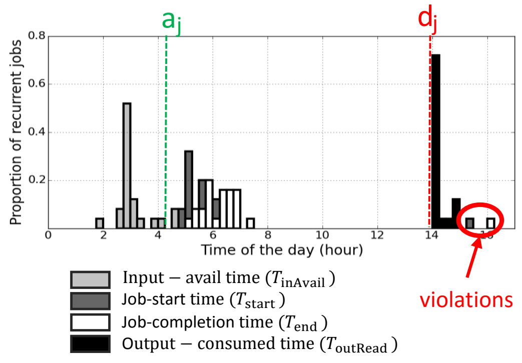


Figure 4: A periodic job from production traces.

(modeled as virtual source/sink nodes). This representation gives us a unique vantage point with nearly perfect close-world knowledge of the meaningful events in the cluster. The PG is constructed by scanning three sets of logs: application logs, filesystem logs, frontend logs. The application logs capture all job-related events such as: start and completion times, failures, and a job's inputs/outputs (this is part of the algebraic representation of the user query). The filesystem logs provide metadata information about files (size, nodes storing each block, etc.). The frontend logs capture upload/download operations from the cluster (i.e., ingress/egress). A daily batch job is used to parse the logs and by means of template-matching extract both the structure and node/edge properties which are then efficiently stored in the PG [37].

Isolating periodic jobs. We group individual job instances in a periodic job, if the templatized job names are an exact match, if source-code signatures are an approximate match, and if submissions have a near-constant inter-arrival time. The latter criterion is evaluated using the coefficient of variation (CV) measure of inter-arrival times. CV is computed as the ratio of median absolute deviation (MAD) (a robust estimate of dispersion) and central value (median), namely $CV = \frac{MAD}{median}$; we filter out jobs with large CV. We derive the period P_j of a job j based on the submission times—not subject to queuing delays.

Estimating SLOs from the PG. With reference to Fig. 4, our goal is to derive estimates for the earliest start time a_j and the deadline d_j for the job. To this end, we rely on four random variables, in chronological order: T_{inAvail}, time at which job inputs are available (i.e., the time of last write to any input); T_{start}, time when the job starts execution; T_{end}, time when the job completes execution; T_{outRead}, time at which any job output is first read. All these times are defined relative to the start of the current period $T_{\text{periodStart}}$[4]. We say that a job has an *actionable deadline* if its output

[4]The start time of the period of the i^{th} job instance is given by $T_{\text{periodStart}} = \text{AbsoluteReferenceTime} + i \cdot \text{period}$, where AbsoluteReferenceTime is the time of first event recorded for the periodic job.

is consumed at an approximately fixed time, relative to the start of the period (e.g., everyday at 4pm), and if there is non-trivial amount of slack between the job end and the deadline. Formally this means imposing thresholds on $CV(T_{\text{outRead}})$ and median $\left(\frac{T_{\text{outRead}}-T_{\text{end}}}{T_{\text{end}}-T_{\text{start}}}\right)$. Finally, a_j and d_j are derived[5] as percentiles of the distributions of T_{inAvail} and T_{outRead} (e.g., 95th and 50th percentiles, respectively). The vast majority of periodic jobs in our workloads have actionable deadlines (§9), and will be offered an inferred SLO. The remainder will continue running with manually provisioned resources.

5 Job Resource Model

The second part of our inference module produces a resource allocation $\mathtt{R}_j^*$ that has high fidelity to the actual requirements of some periodic job j. In a nutshell, *Morpheus* collects resource usage patterns of periodic jobs over N_j instances that have run in the past, and solves an LP that "best fits" all patterns. Fig. 5 shows 4 runs (`R1-R4`) of a TPC-H query that were used, along with other runs, to generate $\mathtt{R}^*$. The underlying optimization is governed by a parameter $\alpha \in [0,1]$ which determines the extent to which one wishes to reduce over-allocation of resources ($\alpha = 1$ hinting the maximal reduction of over-allocation). The usage patterns are captured as a set of skylines, one per run of a periodic job j. The resource allocation $\mathtt{R}_j^*$ is defined as the amount of resources to be provisioned (e.g., number of containers) at any point in time, for the successful execution of the different runs of j. For ease of presentation, we omit the index j, yet recall that all quantities below are for the same periodic job.

To derive the resource allocation, we first align the start times of all the job runs (instances), and quantize time, so that each quantized time-step corresponds to a fixed actual duration (e.g., one minute). Formally, a skyline for the i-th instance can be defined by the sequence $\{s_{i,k}\}$, the average number of containers used for each time-step k ($k \in 1,\ldots,K$). Using a collection of sequences as input, the optimization problem outputs the vector $\mathbf{s} = (s_1,\ldots s_K)$—the number of containers reserved at each time-step.

Our optimization objective is a cost function which is a linear combination of two terms: One term which penalizes for "over-allocation" $A_o(\mathbf{s})$, and another term which penalizes for "under-allocation" $A_u(\mathbf{s})$, both illustrated in Fig. 5; formally, we wish to minimize $\alpha A_o(\mathbf{s}) + (1-\alpha)A_u(\mathbf{s})$. Next we describe these terms.

[5]Note that, for a small fraction of the jobs, the periodicity of the job j can be smaller than the one of its consumers (e.g., daily jobs rolled up in a monthly report). In this case, we force a deadline based on the smallest periodicity to ensure the resource provisioning load is distributed over time (e.g., daily) instead of accumulated at the end (e.g., monthly). We confirmed with users that this aligns with their intents.

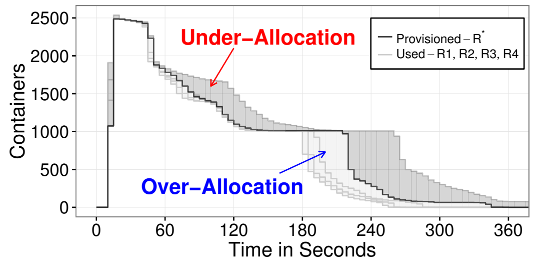


Figure 5: LP deriving a provisioned skyline R*, from four runs (R1-R4) of TPC-H Query12 (10TB scale).

Over-allocation penalty. The over-allocation penalty is defined as the average over-allocation of containers. Formally, the expression $(s_k - s_{i,k})^+ = \max\{s_k - s_{i,k}, 0\}$ is the instantaneous over-allocation for instance i at time-step k. Accordingly, the over-allocation penalty is given by $A_o(\mathbf{s}) = \frac{1}{N}\sum_{i=1}^{N}\sum_k (s_k - s_{i,k})^+$.

Under-allocation penalty. We define a penalty which captures the *eventual* under-allocation of resources. Intuitively, we allow the job to "catch up" on under-allocations using resources available later in the run. Formally, we define the *debt* for instance i at time-step k as $D_{i,k}(s_1,\ldots,s_k) = (D_{i,k-1} + s_{i,k} - s_k)^+$, with $D_{i,0} = 0$. Observe that the allocation can decrease the debt over time, but cannot accumulate "credit" for later times (i.e., the debt cannot go below zero). The under-allocation penalty is the average debt at the last time step. Accordingly, $A_u(\mathbf{s}) = \frac{1}{N}\sum_{i=1}^{N} D_{i,K}(\mathbf{s})$.

The idea behind choosing these particular forms of penalties is to model, as closely as possible, the usage of allocated resources by a job that requests them. Particularly, the over-allocation penalty models the amount of unused resources because the job instance doesn't need them. Wasted resources allocated in a time-step cannot be recovered back at a later time-step. However, a shortage of resources at a time-step can be satisfied at a later point in time assuming the job is elastic. Final shortage of provisioned resources has to be counted only at the end; hence motivating the under-allocation penalty.

Avoiding lazy solutions. Just optimizing the above criteria can lead to solutions that lazily under-provision initially and compensate by aggressively allocating towards the end of a job's execution. So we add the following regularization constraint to the optimization problem $\frac{1}{N}\sum_{i=1}^{N}\frac{\sum_k (s_{i,k}-s_k)^+}{\sum_k s_{i,k}} \le \varepsilon$. In words, we wish to sustain the average normalized instantaneous under-allocation below a threshold ε. While the objective and the constraints have non-linear terms, the optimization problem can be casted as an LP through standard lossless transformations.

The "right" value of ε may depend on the job characteristics (e.g., size, duration). In order to reduce the burden of calibrating the value of ε for every job, we roll ε into the optimization problem as follows. We add a linear term $\beta \cdot \varepsilon$ to the objective function. The value of β is set proportional to the other terms in the objective function, to make it relevant. Specifically, we solve the original optimization problem (without the $\beta \cdot \varepsilon$ term), and obtain a value V. We then set β to be a fraction of that value. Through experiments across many jobs, we found that setting β as $0.1V$ yields good results across the board.

Complexity. The LP has $(O(N \times K))$ number of variables and constraints. Our sampling granularity is typically one minute, and we keep roughly one-month worth of data. This generates less than 100K variables and constraints. A state-of-the-art solver (e.g., Gurobi, CPlex) can solve an LP of millions of variables and constraints in up to few minutes. Since we are way below the computational limit of top solvers, we obtain a solution within few seconds for all periodic jobs in our clusters.

Estimating parallelism. We assume that the skylines used to derive $\mathtt{R}^*$ are generated under capacity allocations sufficient to satisfy the maximum parallelism a job instance can harness. This assumption holds for production jobs because they are typically over-allocated to meet their deadlines. Under this assumption, we treat the estimate $\mathbf{s} = (s_1, \ldots s_K)$ of a job's resource requirement as also being its maximum parallelism for each timestep k. Further, we assume by default that the minimum parallelism of a job is one container (i.e., any requirement s_k can be stretched over time); this assumption can be overridden by either users or operators, assuming that they have additional knowledge about the inner working of the jobs. Inferring the min-parallelism automatically remains an open future direction.

6 Packing multiple periodic jobs

In this section, we provide an overview of *LowCost* – the algorithm we use to pack multiple periodic jobs.

6.1 Periodic reservations

Regardless of the packing algorithm we shall use, we face a practical challenge of how to reserve resources for multiple, possibly infinite, instances of a periodic job. It is inefficient to calculate and store a separate reservation for each instance of a periodic job. To address this challenge, we force the constraint that all instances associated with the same periodic job would have the same reservation across runs (namely, the same offset with respect to the period of the job). E.g., a daily job which requires 10 containers for one hour between 10am and 4pm maybe forced to execute between noon to 1pm every day. While this design choice might reduce the flexibility of a reservation-packing algorithm, it provides stronger predictability to users and reduces allocation complexity.

Having a fixed offset for each periodic jobs produces a repeating pattern in the overall allocation of all periodic jobs. We identify and store the smallest repeating unit which can accurately capture this recurring pattern in the set of all periodic jobs. In particular, we use the Least Common Multiple (LCM) of the time periods as the length of the internal storage unit. This ensures that all periodic jobs align with the boundaries of the storage unit; see Fig. 6 for an illustration. From an algorithmic perspective, one can determine how to pack multiple periodic jobs by only focusing on the LCM representation. This speeds up the packing algorithm, as it does not need to consider separately each instance of the periodic job.

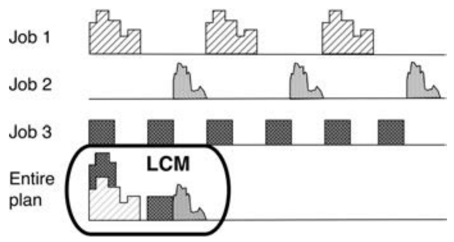


Figure 6: Illustration of LCM representation for multiple periodic reservations.

One may argue that the LCM can get very large, due to slightly "off-kilter" periods of a few jobs (e.g., 58 minute period). However, as shown in Fig. 1b, the distribution of periods in our clusters shows that most period values are divisors of one day. Accordingly, in practice, we set the LCM to be one day. The small fraction of jobs with periods that are not amenable (off-kilter or periods larger than one day) are accommodated using non-periodic reservations for each instance. We note that the LCM can be reconfigured in case of many such outliers.

6.2 Problem formulation

Setting. The input for a planning algorithm is a set of periodic jobs and a time range $[0,T]$, which represents the LCM period as described above. These jobs are typically revealed to the system one by one – i.e., in an online fashion. For simplicity, we describe the algorithmic problem under the assumption that each job has one instance within the LCM; we remove this assumption towards the end of the subsection. Each job j is characterized by a start time a_j, a deadline d_j, and a collection of stages $k \in [1, K_j]$. Each stage k captures a timestep of the reservation (see §5), hence is characterized by a total demand of s_k^j con-

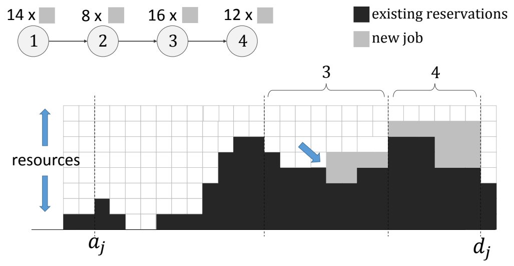


Figure 7: An example of *LowCost* execution. The new job has four stages with different number of containers. Stage 3 is currently being provisioned. Since the stage demands 16 containers and the total remaining demand is 38 containers, the time-interval for this stage is 16/38 of the time available, i.e., $\frac{16}{38} \cdot 19 = 8$. The arrow indicates where the next container of stage 3 would be allocated.

tainers; the stage may also have a minimum parallelism constraint (or gang size) of g_k containers.

Objective and Constraints. The goals of the packing algorithm are to (i) allocate containers to all periodic jobs, such that their requirements are met by the deadline, and (ii) minimize the waiting time for non-periodic, ad-hoc jobs. These goals can be better fulfilled if the cluster load is *balanced* over time. Intuitively, a balanced allocation increases the likelihood of accommodating future jobs (both periodic and non-periodic) that arrive into the system. As a concrete measure for a balanced allocation, the objective of *LowCost* is to minimize the maximal total allocation over time. We impose the following constraints on any solution. First, unless strictly necessary, we do not allow re-scheduling of jobs that are already in the system. This is important for business continuity. Second, because we typically use a sequence of stages to represent resource skylines, the entire allocation has to be *contiguous*, i.e., we do not allow "holes" in the allocation.

We note that the resulting online scheduling problem is hard already for single-stage jobs – Even the offline problem is NP-hard, as it generalizes the makespan minimization problem on multiple machines (e.g., [34]).

Requirements. We highlight the main requirements from a packing algorithm. The offline version of our planning problem can be casted as a Mixed Integer Linear Program (MILP). However, we prefer a quicker and "lighter" solution in terms of the running complexity. The main reason for not relying on rather costly solvers, is that *Morpheus* may often update the reservation plan. For example, upon arrival of a new periodic job, or as a consequence of changes in the resource estimations (hence reservation) of a job. On a related note, we need an *incremental* solution. That is, we wish to keep the reservations steady for jobs that are already in the system, and do not exhibit substantial changes in their resource demand.

6.3 Packing with *LowCost*

Cost function. *LowCost* uses a *cost-based* approach for allocation of containers that takes into account current cluster allocation and the resource demand of each job – each time slot t is associated with a cost $c(t)$. By default, the cost function $c : \mathbb{N} \rightarrow \mathbb{R}$ represents the current load of the cluster. Formally, $c(t) = \max\left\{\frac{\text{load}(MEM,t)}{\text{capacity}(MEM,t)}, \frac{\text{load}(cores,t)}{\text{capacity}(cores,t)}\right\}$, where load$(\cdot,t)$ represents the total allocation of the resource at time t, and capacity$(\cdot,t)$ represents its capacity.

The basic algorithm. In a nutshell, the idea behind *LowCost* is to allocate each incoming job in a way that is cost-efficient with respect to $\max_t c(t)$. To that end, *LowCost* follows a greedy procedure which places containers iteratively at cost-efficient positions.

In more detail, *LowCost* handles the stages one by one in reverse chronological order. For each stage k, *LowCost* first sets a time interval $I_{j,k} = [\tau^l_{j,k}, \tau^r_{j,k}]$ during which the stage can be allocated. $\tau^r_{j,k}$ is set right before the allocation of stage $k+1$. The length of $I_{j,k}$ is set proportional to the ratio between the demand of the stage and the total demand of the remaining stages, i.e., $\frac{s^j_k}{\sum_{k'=1}^{k} s^j_{k'}}$; see Fig. 7 for an example. To accommodate the contiguous allocation constraint, the eligible time steps for allocating the next gang of a given stage are $[\tau^{\text{cur}}_{j,k} - 1, \tau^r_{j,k}]$, where $\tau^{\text{cur}}_{j,k}$ is the leftmost timestep which includes some non-zero value for the current allocation to the stage. *LowCost* repeats the above procedure for different end points, and chooses the allocation with the minimum cost.

Multiple instances. Finally, a periodic job may have *multiple instances* within the LCM (e.g., an hourly job j, where the LCM is one day). As mentioned earlier, we place all the instances of the job with the same offset with respect to the period of the job (e.g., all instances of j should start at the same time-of-day). We incorporate this constraint in *LowCost* as follows. Observe that we essentially need to decide on the placement of a single instance. To do so, for each timestep within the job's period, we set the cost as the maximal cost across all timesteps with the same offset with respect to the period. For example, the cost seen by j at the 5-th minute would be the maximum over the costs at 12:05, 1:05, etc. *LowCost* then places a single instance based on these costs, and repeats the assignment for all instances within the LCM.

We wish to analyze in isolation the consequences of this choice. Accordingly, for the analysis sake, we assume that all periodic jobs have the same skyline requirement for their instances, but still jobs can differ in their period. Under these assumptions, we measure the performance of *LowCost* using the standard measure of competitive ratio. The competitive ratio of an online algorithm is the ratio of cost (for our problem, the maximal height of the allocation) incurred by the online algorithm compared to the offline optimal solution. Let $P_{\min}, P_{\max}$ denote the minimum and maximum period of jobs within the LCM. We have the following guarantee:

Theorem 6.1 *Under the above assumptions,* LowCost *is* $O\left(\log\left(\frac{P_{\max}}{P_{\min}}\right)\right)$*-competitive for the objective of minimizing the maximum height of the allocation.*

The proof follows by showing that *LowCost* leads to an efficient (constant-competitive) schedule when jobs have the same period. The log-factor arises due to the range of possible job periods. Intuitively, this result implies that there is bounded performance loss due to the combination of our design and algorithmic choices.

Non-periodic jobs. So far we described how we reserve resources for periodic jobs. We now briefly address how *Morpheus* handles non-periodic jobs. The design of *Morpheus* assumes that periodic jobs have strict priority over non-periodic (mostly ad-hoc) jobs. This is commensurate with our analysis, which indicates that the bulk of periodic jobs are (business-critical) production jobs (see §2). Accordingly, when *Morpheus* needs to allocate resources to a new periodic job, it ignores most of the scheduled non-periodic jobs (excluding periodic jobs that are handled as non-periodic ones), and then attempts to reallocate resources for non-periodic jobs in case they need more resources. Specifically, *Morpheus* places the non-periodic using the same logic of the basic *LowCost* algorithm, described above. The only difference is that the plan for the lower-priority non-periodic jobs uses the residual capacity, after subtracting the chunk used for periodic jobs.

7 Dynamic Reprovisioning

While reservations can eliminate sharing-induced unpredictability, they provide little protection against inherent unpredictability arising from hard-to-control exogenous causes, such as *infrastructure* issues (e.g., hardware replacements (see §2), lack of isolation among tasks of multiple jobs, and framework code updates) and *job-centric* issues (changes in the size, skew, availability of input data, changes in code/functionalities, etc.).

Although eliminating all the causes of unpredictability is very hard, we can mitigate their impact on SLO attainment during runtime, by dynamically modifying the current instance of a periodic reservation. To that end, we design a *dynamic* reprovisioning mechanism which is triggered when a job execution appears to be headed for an SLO violation.

Dynamic Reprovisioning Algorithm (DRA). The Dynamic Reprovisioning Algorithm (DRA) we currently employ in *Morpheus* continuously monitors the resource consumption of the job, compares it with the resources allocated in the reservation and intuitively "stretches" the skyline of resources to accommodate a slower-than-expected job execution. Reprovisioning is triggered when a job resource demand (used containers plus pending ask) exceeds the resources allocated in the skyline. Extra resources are granted for up to T seconds (default 1min), after which DRA is reevaluated again. The amount of extra resources is based on the job's instantaneous demand, but capped at $\rho * \max(R_{\text{recent}})$ where R_{recent} is the amount of resources allocated in the skyline in the last few minutes (default 2min), and ρ is a fudge factor (default value 2) that allows an elastic job to use extra parallelism to make up for lost time; note that DRA verifies that the job does not get more resources than it requests. Given this proposed reprovisioning, DRA updates the current instance of the periodic reservation (by increasing it locally). This is done by invoking *LowCost*, which ensures the update is accepted only if enough resources exist in the plan.

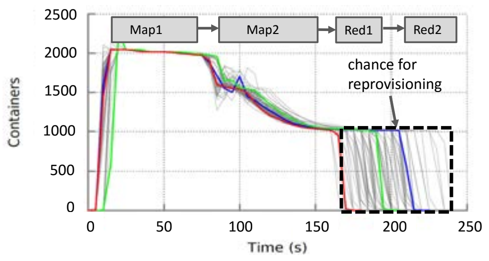


Figure 8: Resource consumption over time for 100 runs of TPC-H Query1 on 2200 parallel containers (job running alone in the cluster).

The proposed heuristics cope well with the inherent unpredictability we observed in Section 2.2. We show this by plotting in Fig. 8 the resource consumption over time for the TPC-H Query1 (100 runs, and highlighting 3 random ones in red/green/blue). DRA kicks in for jobs that have straggling Map2 tasks, which translates in a delayed start of Red1 stage. By extending the 1000 containers allocation at the end of Red1 by an extra minute we allow most jobs to complete effectively. Similar analysis has been performed for other TPC-H queries with equally good results, and in §9 we validate DRA performance on large production traces.

DRA is simple to implement and rather robust, however deeper understanding of the application-framework could lead to more precise reprovisioning decisions. In

Morpheus' pluggable architecture, this could be achieved by borrowing techniques from [15, 41, 19].

Adjusting *LowCost* to facilitate reprovisioning. The position of the original reservation allocation with respect to the $[T_{\text{inAvail}}, T_{\text{outRead}}]$ window is critical for the effectiveness of dynamic reprovisioning. In order to improve the success probability of reprovisioning, it is necessary to allocate resources far away from the deadline (allowing sufficient slack in time for the reprovisioning algorithm to compensate a slower than expected run). However, in case of high variance in input data availability, it is beneficial to place the allocation close to the deadline (to ensure that data is available before allocation and thus reduce the probability of reprovisioning). To account for this trade-off, we adjust *LowCost*'s cost function for "problematic" jobs (e.g., jobs with high CV for $T_{\text{start}} - T_{\text{inAvail}}$, $T_{\text{outRead}} - T_{\text{end}}$) by adding an *alignment penalty* . Specifically, the penalty is linearly proportional to the absolute time-distance between the mid-point of the allocation, and the mid-point between the start time and the deadline (i.e., $\frac{a_j+d_j}{2}$). This penalty incentivizes allocations that are not too close to neither the start or the deadline of the job. This trades the two dangers of allocating resources before the input is available, and not having enough slack after the allocation before the deadline.

8 Implementation

We implement the design of §3 as extensions to Apache Hadoop / YARN [51]. Referring back to the architecture of Fig. 3, we implement the three components of *Morpheus* as follows. First, the automatic inference engine operates as a standalone service. It continuously consumes provenance and telemetry data and submits reservation requests to the Resource Manager (RM)—YARN's centralized scheduler component [51]—via its REST endpoint. Second, the reservation placement component implements *LowCost* as in-process functionality of the RM. Third, the dynamic reprovisioning mechanism is implemented as a monitoring thread in the RM, which observes job resource requests and triggers resizing of reservations. Each of the above components is highly pluggable and can easily be specialized to leverage framework-specific knowledge, such as [19, 41, 15].

In the rest of this section, we discuss some of the engineering challenges in building a production-ready system.

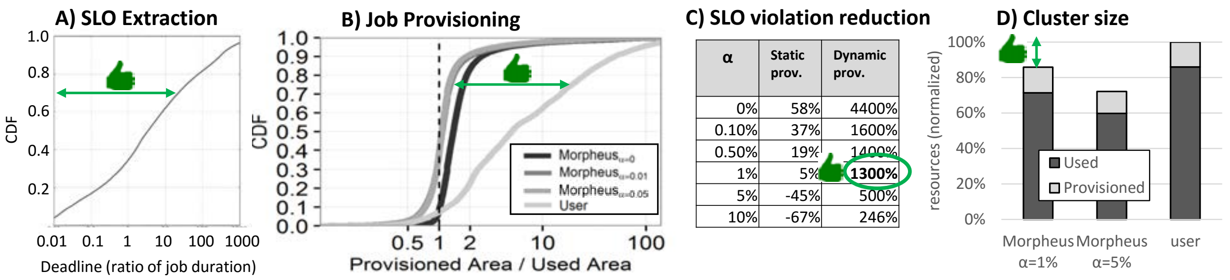
Scalability. *Morpheus*' periodic reservations are instantiated as per-job queues in the RM. Each queue's capacity continuously grows and shrinks according to the provisioning skyline. YARN's RM scheduler [51], is designed to support a small number of infrequently reconfigured queues (e.g., one per division of a company). Hence, the implementation leveraged strict consistency via locking for queue updates. This limited *Morpheus*' scalability to levels far below our production needs. We address this by substantially reworking the RM scheduler locking mechanisms through a combination of finer-granularity locking and lock-free data structures. The key intuition is that the RM operates as an asynchronous event-driven system based on heartbeats and, therefore, is amenable to operating with relaxed consistency. We carefully study the effects of our changes and confirm that they induce very small and transient inconsistencies, that are naturally resolved without visible impact within milliseconds. This results in a sustained scalability orders of magnitude higher than the baseline. We showcase this experimentally running on a 2700-node cluster in §9.3.

Cold-Start. An obvious concern for a system that relies on history to make inference is how to handle cold-start scenarios, such as non-recurring jobs or initial runs of a new recurring job. We have three lines of defense to cope with this problem: *(a)Backward compatibility:* Our approach by design is able to support running jobs with existing fair-queueing infrastructure mode. *(b) Manual SLOs and job resource models:* The APIs supporting the sign-off (step 3) in our job lifecycle can be used to supply a manually defined SLO and job resource model (both for periodic and non-periodic jobs [13]). *(c) Application-specific tools:* Given a fixed application framework (e.g., Hive/Giraph/Scope/Spark) it is possible to build tools that leverage sample runs and careful modeling to predict the behavior of the full-scale execution of the jobs. We experimentally integrated with Predict [41] to support Giraph computations, as well as recently enabled similar functionalities for Hive/Tez/MapReduce with the Perforator [15] effort. In [15], we take Hive queries and perform cardinality estimation via lightweight profiling of UDFs. We then use this accurate cardinality estimates together with explicit models of Tez/MapReduce pipelining and parallelism and hardware performance profiles to estimate a job demand model. Perforator is integrated with our infrastructure, but complete integration between Morpheus and Perforator technologies is part of our future work.

9 Experimental Evaluation

In this section, we demonstrate effectiveness and scalability of *Morpheus* through simulations and cluster runs.

Experimental Settings Our experiments are based on two production traces and a synthetic benchmarking suite: *Enterprise-trace*, a one-month trace of jobs running on a large 50k-node production *COSMOS* cluster [9]; *Hadoop-trace*, a three-month trace derived from a 4k nodes production Hadoop cluster; *TPC-H*, the standard TPC-H



α	Static prov.	Dynamic prov.
0%	58%	4400%
0.10%	37%	1600%
0.50%	19%	1400%
1%	5%	**1300%**
5%	-45%	500%
10%	-67%	246%

Figure 9: Comparison of *Morpheus* with current user manual provisioning.

benchmark running on Hive/Tez at 10TB scale. The enterprise-trace has been discussed in §2, and TPC-H is well documented [12]. The table below presents a breakdown of jobs types and size for our hadoop-trace. Jobs are clustered into multiple classes based on duration and size.

Hadoop-trace

framework	class	freq. %	avg duration (sec)	avg parall.
MR/TEZ	S	7%	73	1.5
	M	15%	156	19
	L	0.6%	2778	469
SPARK	S	39.8%	173	2.6
	M	14.52%	605	18
	L	7.8%	1400	88
	XL	4%	6300	510
	XXL	8.6%	24570	1000
MPI	-	1.56%	7800	400

For each category we extract salient statistical distributions: job arrival times, workload frequency, job parallelism, and job duration. These distributions are used to power a Gridmix-based [48] load generator.

9.1 Performance on the enterprise-trace

First, we challenge *Morpheus* in a simulation based on our largest dataset, the **enterprise-trace**. For this data-set we have full provenance graph (PG) and telemetry information, and we can thus test all components of *Morpheus*.

Sensible SLOs for most jobs. A pressing question we want to answer is whether the SLOs we derive are representative of user expectations. Short of a full-scale user study, we study a reliable proxy metrics: job success/-failure. Given job pairs $A \rightarrow B$, such that B is the first consumer of A's output, we measure from the trace:

$P(B_{\text{fail}} \mid A_{\text{missSLO}}) \approx P(B_{\text{fail}} \mid A_{\text{fail}}) > 4 \times P(B_{\text{fail}} \mid A_{\text{meetSLO}})$

This shows that the negative impact of missing a deadline is comparable with the impact of complete failure of the job. This is empirically $4\times$ worse for the dependent job B than if A had met the SLO.

Second, we observe that *Morpheus* SLO target extractor successfully derives SLOs for over 70% of the millions of instances of periodic jobs in the enterprise-trace. For the remainder we have too little data in our trace to derive SLOs with good confidence (e.g., we only have four samples in our trace for jobs with weekly periodicity).

SLOs, job modeling, packing, and reprovisioning. In Fig. 9 we use our enterprise-trace (70% training and 30% testing) to show *Morpheus*' ability to: (A) extract SLOs, (B) derive job resource models, (C) achieve high SLO attainment gains over the baseline, and (D) pack reservation efficiently (measured as potential cluster reduction). In Fig. 9a, we present a CDF of the ratio between the slack (time between job-completion and deadline) and the job duration ($\frac{T_{\text{outRead}}-T_{\text{end}}}{T_{\text{end}}-T_{\text{start}}}$). The majority of jobs have substantial amount of slack (almost 70% of jobs have enough slack to serially execute two or more times before the deadline)—this flexibility is leveraged during packing. Fig. 9b compares the job provisioning achieved by *Morpheus* under different assignments of the parameter α with the user-supplied one (matching our motivation Fig. 2a). *Morpheus* drastically outperforms the user, by being consistently closer to the ideal provisioning (1:1 ratio, shown as vertical dotted line). Different assignment of α affect how tightly the skyline is fitted, but also how likely we are to miss an SLO (Fig. 9c). We find that a value of 1% leads to the best balance, yielding $13\times$ reduction of the worst-case SLO misses—these are defined as the amount of SLO violations a periodic job would incur if no opportunistic (fair-share) capacity is available. Finally Fig. 9d shows that our packing algorithms manage to handle the complex skylines produced by the job modeling component, and leverage the slack in SLOs to densely pack the cluster agenda. This matches our important constraint of not increasing the cluster cost (but actually lowering it). The ratio between used and provisioned indicates that we achieve high-utilization, even though we rely solely on guaranteed provisioning, while the user compensate with under-allocation via opportunistic fair-sharing. Note that our unused capacity is anyway redistributed fairly, but we do not rely on it to achieve high utilization.

9.2 Breakdown of contributions

Fig. 10a shows a breakdown of contributions of our static techniques. We fix a target SLO attainment level $5\times$

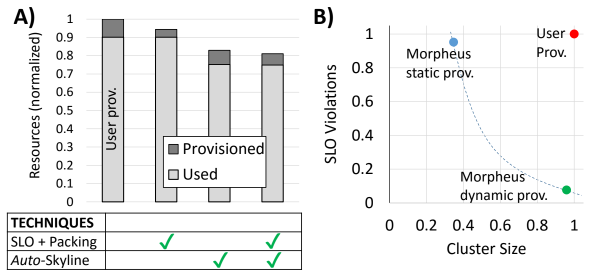


Figure 10: Gain break-down: each technique employed by *Morpheus* delivers sizable improvements.

above the baseline, and show the smallest size cluster required to achieve that under different combinations of our techniques. In particular, we show that: 1) SLO-extraction + packing lower the baseline cluster size by 6%, 2) Job resource modeling, i.e., using our skyline instead of user supplied provisioning, can alone lower cluster size by 16%, and that 3) when combined they achieve 19% total reduction. Fig. 10b highlights the trade-off between utilization and predictability, by showing how turning on/off our dynamic reprovisioning we can either: 1) match the user utilization level, and deliver 13× lower violations, or 2) match the current SLO attainment and reduce cluster size by over 60% (since we allow more aggressive tuning of the LP, and repair underallocations dynamically). Hence, each of the techniques we developed is required and supplies a substantial portion of our overall win.

9.3 Physical deployments and scale tests

In this section, we challenge *Morpheus* with a combination of the Hadoop-trace and the TPC-H workloads. We test our system under 3 environments: (a) 275-node cluster with 2200 containers (1core, 8GB RAM per container), (b) 2700-node cluster (100k containers) and (c) 4000-node production cluster.

Adversarial workload. We validate *Morpheus*' ability to protect jobs from an adversarial workload. In Fig. 12, we show an hourly periodic workload comprised of several TPC-H queries running on a 275-node cluster (equivalent to 2200 containers). The workload imposes heavy load on the cluster, with container utilization hovering near 100% of the available capacity during most of the experiment. The jobs are submitted periodically within a reservation we derived from historical runs. We then surround the periodic jobs with thousands of ad-hoc jobs from the Hadoop-trace which can take upto 75% of the cluster capacity. Thus, periodic reservations are run against a cluster stressed with production workload. *Morpheus* successfully eliminates all sharing-induced variability. To further challenge our system, we manually delay the start of one of the queries. The system immediately reacts by dynamically reprovisioning the (delayed) job with extra capacity, compensating for our actions and meeting the target SLO.

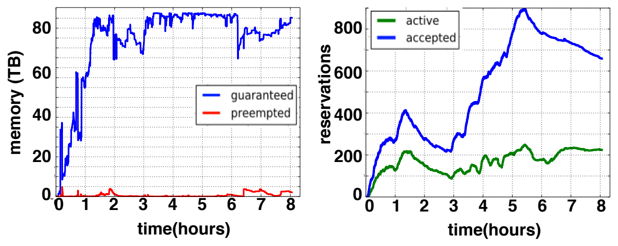


Figure 11: Scalability metrics for large scale *real cluster* run (on 2700 nodes)

Scale test (2700 nodes). We validate *Morpheus*' scalability to target production clusters, by running it live on a *2700-node cluster*, scheduling almost 100k concurrent containers through the ResourceManager. This is a high-load workload designed to stress the scheduling infrastructure. We run a sustained 8hr experiment, with hundreds of reservation submissions per hour. We measure the system performance both as perceived by the user (not shown), and as observed by instrumented system components (Fig. 11). The key takeaway of this experiment is three-fold: 1) we demonstrate that *Morpheus* is able to sustain high load on a large cluster, 2) we confirm that in a real deployment *Morpheus* can achieve high plan utilization, 3) we confirm that user-facing latencies are in-line with production cluster user expectations. We see up to 900 concurrent reservations in the plan, with up to 270 of them active throughout the 8hr run. At peak, aggregate guaranteed capacity exceeds the 92TB of container memory, reaching maximum cluster capacity. The system remains responsive throughout the experiment with reservation submission latencies within 10sec.

Production deployment. We validate our system by deploying it in a 4000-node production environment. In this context, we are only allowed to run a small number of periodic jobs via reservations, while the bulk of the load is imposed by ad-hoc and manually provisioned jobs. Focusing on a periodic run of TPC-H Query3, the runtime variability was well controlled, despite utilization swings of whole cluster in excess of 69k cores during the job execution. During the same period, jobs running without the protection of reservations observed much larger variance.

10 Related Work

SLO extraction. To the best of our knowledge, we are the first to propose fully automated extraction of SLOs from historical data. Close related work focused on semi-automated, iterative generation of SLOs for databases [40] and web services [46].

Runtime/provisioning estimation. Substantial re-

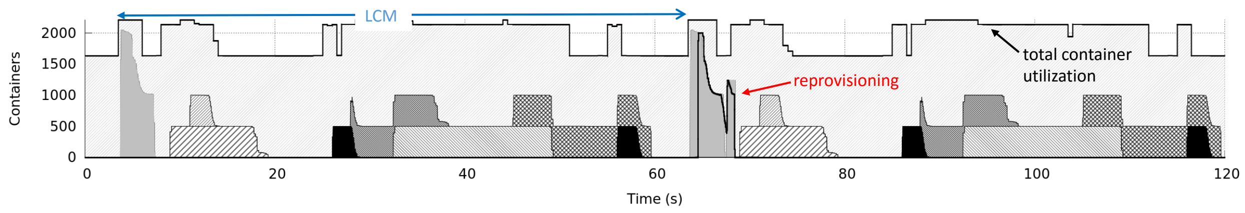


Figure 12: Run on 275-node cluster: shows an example of successful dynamic-reprovisioning.

lated work has been devoted both in database and systems literature to estimate query runtimes, and resource needs. Runtime prediction has been studied in databases [33, 10, 21], Big-Data/Cloud [39, 38, 52, 20, 17], and HPC/Grid computing settings [53, 31, 45]. A large body of work [54, 32, 11] leveraged known MapReduce job structure to accurately predict both resource demand and runtimes across different data input sizes. Our architecture allows using any of these techniques, when the application framework is known, while this paper presents a framework-agnostic solution purely based on history. History-based modeling has been used in other contexts: failure-prediction for quality of service (QoS) [18], and resource allocation in business process management [28, 3, 35].

SLO enforcement. Automatic techniques for meeting SLOs [59, 19], use a combination of profiling and job structure knowledge for runtime prediction. PRESS [23] focuses on meeting SLOs at a single-node level and can adjust allocated resources online. Jockey [19] provides a solution for dynamic reprovisioning based on job models derived from execution history and job's internal dependencies. It can be used as a framework-specific dynamic reprovisioning policy. *Morpheus* provides deadlines and global arbitration, which are beyond the scope of [19]. Other dynamic enforcement mechanisms include control-theoretic approaches such as [16, 49].

Online packing and scheduling. The scheduling problem solved by *Morpheus* is a significant generalization of online multidimensional bin-packing problems [4, 25, 7, 5] and online deadline-scheduling problems (see [36, 6] and references therein). Placement in periodic settings has also been studied in the context of real-time and multi-processor machines [8, 47, 14]. However, the combination of jobs with stage-dependencies, periodicity and and deadlines requires novel algorithm design.

Cluster Scheduling. There has been a substantial body of work on cluster scheduling for big-data analytics [22, 29, 58, 50, 24]. Corral [30] leverages job recurrence and predictable resource requirements to coordinate data and task placement for higher utilization, but does not consider SLOs. Based on published material, SLO inference/enforcement is not present in Mesos [27], Borg [55], and Omega [44]. However, *Morpheus*' mechanisms can be adapted to alternative underlying schedulers. Apollo [9] makes more explicit trade-offs on time vs locality at the task level, but does not provide job-completion SLOs. YARN's reservation system [13] serves as a base for *Morpheus*, but it left unsolved the SLO and job resource model derivation, support for periodic reservations, and dynamic reprovisioning. Moreover, the packing algorithms we present here outperform the one in [13] even for non-periodic jobs [1].

11 Conclusion

In this paper, we present *Morpheus*, a system designed to resolve the tension between predictability and utilization—that we discovered thorough analysis of cluster workloads and operator/user dynamics. *Morpheus* builds on three key ideas: automatically deriving SLOs and job resource models from historical data, relying on recurrent reservations and packing algorithms to enforce SLOs, and dynamic reprovisioning to mitigate inherent execution variance. We validate our design and implementation against large production traces, and on a 2700-node cluster. *Morpheus* reduces worst-case SLO violations by 5-13×, while concurrently reducing the cluster footprint by 14-28%. Overall, *Morpheus* enables predictable performance with less resource provisioning—a win-win for operators and users.

Acknowledgements

We thank our shepherd Sasha Fedorova, and the reviewers for their insightful feedback. We are particularly indebted to Chris Douglas for numerous conversations that helped shape this project. We are also grateful to many colleagues for many great discussions: Ricardo Bianchini, Roni Burd, Kishore Chaliparambil, Chris Douglas, Avrilia Floratou, Giovanni M. Fumarola, Greg Ganger, Brighten Godfrey, Solom Heddaya, Virajith Jalaparti, Alekh Jindal, Srikanth Kandula, Konstantinos Karanasos, Alica Li, Joseph Naor, Vivek Narasayya, Sekhar Pasupuleti, Sean Po, Raghu Ramakrishnan, Keerthi Selvaraj, Arun Suresh, Vin Wang, and Markus Weimer.

References

[1] LowCost: A Cost-Based Placement Agent for YARN Reservations. `https://issues.apache.org/jira/browse/YARN-3656`.

[2] Support for recurring reservations in the YARN Reservation System. `https://issues.apache.org/jira/browse/YARN-5326`.

[3] M. Arias, E. Rojas, J. Munoz-Gama, and M. Sepúlveda. A framework for recommending resource allocation based on process mining. In *Business Process Management Workshops - BPM 2015, 13th International Workshops, Innsbruck, Austria, August 31 - September 3, 2015, Revised Papers*, pages 458–470, 2015.

[4] J. Augustine, S. Banerjee, and S. Irani. Strip Packing with Precedence Constraints and Strip Packing with Release Times. *Theoretical Computer Science*, 410(38-40), 2009.

[5] Y. Azar, I. R. Cohen, and I. Gamzu. The loss of serving in the dark. In *Proceedings of the Symposium on Theory of Computing Conference*, STOC, 2013.

[6] Y. Azar, I. Kalp-Shaltiel, B. Lucier, I. Menache, J. S. Naor, and J. Yaniv. Truthful online scheduling with commitments. In *Proceedings of the Sixteenth ACM Conference on Economics and Computation*, pages 715–732. ACM, 2015.

[7] N. Bansal and A. Khan. Improved Approximation Algorithm for Two-Dimensional Bin Packing. In *Proceedings of the Annual ACM-SIAM Symposium on Discrete Algorithms*, SODA, 2014.

[8] S. K. Baruah, J. Gehrke, and C. G. Plaxton. Fast scheduling of periodic tasks on multiple resources. In *Proceedings of the 9th International Symposium on Parallel Processing*, IPPS '95, pages 280–288, Washington, DC, USA, 1995. IEEE Computer Society.

[9] E. Boutin, J. Ekanayake, W. Lin, B. Shi, J. Zhou, Z. Qian, M. Wu, and L. Zhou. Apollo: Scalable and coordinated scheduling for cloud-scale computing. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*, pages 285–300, Broomfield, CO, Oct. 2014. USENIX Association.

[10] S. Chaudhuri and V. Narasayya. Self-tuning database systems: A decade of progress. In *Proceedings of the 33rd International Conference on Very Large Data Bases*, VLDB '07, pages 3–14. VLDB Endowment, 2007.

[11] L. Cherkasova. Performance modeling in Mapreduce environments: Challenges and opportunities. In *Proceedings of the 2nd ACM/SPEC International Conference on Performance Engineering*, ICPE '11, pages 5–6, New York, NY, USA, 2011. ACM.

[12] T. P. P. Council. TPC-H benchmark specification. *Published at http://www. tcp. org/hspec. html*, 2008.

[13] C. Curino, D. E. Difallah, C. Douglas, S. Krishnan, R. Ramakrishnan, and S. Rao. Reservation-based scheduling: If you're late don't blame us! In *Proceedings of the ACM Symposium on Cloud Computing*, SOCC, 2014.

[14] R. I. Davis and A. Burns. A survey of hard real-time scheduling for multiprocessor systems. *ACM Computing Surveys (CSUR)*, 43(4):35, 2011.

[15] A. Desai, K. Rajan, and K. Vaswani. Critical path based performance models for distributed queries. In *Microsoft Tech-Report: MSR-TR-2012-121*, 2012.

[16] Y. Diao, J. L. Hellerstein, S. Member, S. Parekh, S. Member, R. Griffith, G. E. Kaiser, S. Member, and D. Phung. A control theory foundation for self-managing computing systems. *IEEE journal*, 23:2213–2222, 2005.

[17] A. V. Do, J. Chen, C. Wang, Y. C. Lee, A. Y. Zomaya, and B. B. Zhou. Profiling applications for virtual machine placement in clouds. In *Cloud Computing (CLOUD), 2011 IEEE International Conference on*, pages 660–667. IEEE, 2011.

[18] J. Ejarque, A. Micsik, R. Sirvent, P. Pallinger, L. Kovacs, and R. M. Badia. Semantic resource allocation with historical data based predictions. In *The First International Conference on Cloud Computing, GRIDs, and Virtualization, CLOUD COMPUTING 2010*, 2010.

[19] A. D. Ferguson, P. Bodik, S. Kandula, E. Boutin, and R. Fonseca. Jockey: guaranteed job latency in data parallel clusters. In *Proceedings of the ACM European Conference on Computer Systems*, EuroSys, 2012.

[20] A. Ganapathi, Y. Chen, A. Fox, R. Katz, and D. Patterson. Statistics-driven workload modeling for the cloud. In *Data Engineering Workshops (ICDEW), 2010 IEEE 26th International Conference on*, pages 87–92. IEEE, 2010.

[21] A. Ganapathi, H. Kuno, U. Dayal, J. L. Wiener, A. Fox, M. Jordan, and D. Patterson. Predicting multiple metrics for queries: Better decisions enabled by machine learning. In *Data Engineering, 2009. ICDE'09. IEEE 25th International Conference on*, pages 592–603. IEEE, 2009.

[22] A. Ghodsi, M. Zaharia, B. Hindman, A. Konwinski, S. Shenker, and I. Stoica. Dominant resource fairness: Fair allocation of multiple resource types. In *Proceedings of the 8th USENIX Conference on Networked Systems Design and Implementation*, NSDI'11, pages 323–336, Berkeley, CA, USA, 2011. USENIX Association.

[23] Z. Gong, X. Gu, and J. Wilkes. PRESS: PRedictive Elastic ReSource Scaling for cloud systems. In *2010 International Conference on Network and Service Management*, pages 9–16, Oct 2010.

[24] R. Grandl, G. Ananthanarayanan, S. Kandula, S. Rao, and A. Akella. Multi-resource packing for cluster schedulers. In *ACM SIGCOMM Computer Communication Review*, volume 44, pages 455–466. ACM, 2014.

[25] R. Harren and W. Kern. Improved Lower Bound for Online Strip Packing. *Theory of Computing Systems*, 56(1), 2015.

[26] H. Herodotou, H. Lim, G. Luo, N. Borisov, L. Dong, F. B. Cetin, and S. Babu. Starfish: A self-tuning system for big data analytics. In *CIDR*, volume 11, pages 261–272, 2011.

[27] 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, pages 295–308, Berkeley, CA, USA, 2011. USENIX Association.

[28] Z. Huang, W. Aalst, X. Lu, and H. Duan. Reinforcement Learning Based Resource Allocation in Business Process Management. *Data and Knowledge Engineering*, 70(1):127 –145, 2011.

[29] M. Isard, V. Prabhakaran, J. Currey, U. Wieder, K. Talwar, and A. Goldberg. Quincy: fair scheduling for distributed computing clusters. In *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*, pages 261–276. ACM, 2009.

[30] V. Jalaparti, P. Bodik, I. Menache, S. Rao, K. Makarychev, and M. Caesar. Network-aware scheduling for data-parallel jobs: Plan when you can. In *Proceedings of the 2015 ACM Conference on Special Interest Group on Data Communication*, SIGCOMM '15, pages 407–420, New York, NY, USA, 2015. ACM.

[31] S. Krishnaswamy, S. W. Loke, and A. Zaslavsky. Estimating computation times of data-intensive applications. *IEEE Distributed Systems Online*, 5(4), April 2004.

[32] P. Lama and X. Zhou. Aroma: Automated resource allocation and configuration of mapreduce environment in the cloud. In *Proceedings of the 9th International Conference on Autonomic Computing*, ICAC '12, pages 63–72, New York, NY, USA, 2012. ACM.

[33] K. Lee, A. C. Konig, V. Narasayya, B. Ding, S. Chaudhuri, B. Ellwein, A. Eksarevskiy, M. Kohli, J. Wyant, P. Prakash, R. Nehme, J. Li, and J. Naughton. Operator and query progress estimation in microsoft sql server live query statistics. In *Proceedings of the ACM SIGMOD International Conference on Management of Data (SIGMOD 2016)*. ACM Association for Computing Machinery, June 2016.

[34] J. K. Lenstra, D. B. Shmoys, and É. Tardos. Approximation algorithms for scheduling unrelated parallel machines. *Mathematical programming*, 46(1-3):259–271, 1990.

[35] T. Liu, Y. Cheng, and Z. Ni. Mining event logs to support workflow resource allocation. *Knowl.-Based Syst.*, 35:320–331, 2012.

[36] B. Lucier, I. Menache, J. S. Naor, and J. Yaniv. Efficient online scheduling for deadline-sensitive jobs. In *Proceedings of the twenty-fifth annual ACM symposium on Parallelism in algorithms and architectures*, pages 305–314. ACM, 2013.

[37] R. Mavlyutov, C. Curino, B. Asipov, and P. Cudre-Mauroux. Dependency-Driven Analytics: a Compass for Uncharted Data Oceans, 2016. Microsoft Technical Report MS-TR-2016-69, `http://bit.ly/2dQfRhc`.

[38] K. Morton, M. Balazinska, and D. Grossman. Paratimer: a progress indicator for mapreduce dags. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of data*, pages 507–518. ACM, 2010.

[39] K. Morton, A. Friesen, M. Balazinska, and D. Grossman. Estimating the progress of mapreduce pipelines. In *Data Engineering (ICDE), 2010 IEEE 26th International Conference on*, pages 681–684. IEEE, 2010.

[40] J. Ortiz, V. T. de Almeida, and M. Balazinska. Changing the face of database cloud services with personalized service level agreements. In *CIDR*, 2015.

[41] A. D. Popescu, A. Balmin, V. Ercegovac, and A. Ailamaki. PREDIcT: Towards predicting the runtime of large scale iterative analytics. *Proceedings of the VLDB Endowment*, 6(14):1678–1689, 2013.

[42] A. D. Popescu, V. Ercegovac, A. Balmin, M. Branco, and A. Ailamaki. Same queries, different data: Can we predict runtime performance? In *Data Engineering Workshops (ICDEW), 2012 IEEE 28th International Conference on*, pages 275–280. IEEE, 2012.

[43] J. Rasley, K. Karanasos, S. Kandula, R. Fonseca, M. Vojnovic, and S. Rao. Efficient queue management for cluster scheduling. In *Proceedings of the Eleventh European Conference on Computer Systems*, page 36. ACM, 2016.

[44] M. Schwarzkopf, A. Konwinski, M. Abd-El-Malek, and J. Wilkes. Omega: flexible, scalable schedulers for large compute clusters. In *Proceedings of the ACM European Conference on Computer Systems*, EuroSys, 2013.

[45] O. Sonmez, N. Yigitbasi, A. Iosup, and D. Epema. Trace-based evaluation of job runtime and queue wait time predictions in grids. In *Proceedings of the 18th ACM International Symposium on High Performance Distributed Computing*, HPDC '09, pages 111–120, New York, NY, USA, 2009. ACM.

[46] J. Spillner and A. Schill. Dynamic SLA template adjustments based on service property monitoring. In *Cloud Computing, 2009. CLOUD'09. IEEE International Conference on*, pages 183–189. IEEE, 2009.

[47] A. Srinivasan and S. Baruah. Deadline-based scheduling of periodic task systems on multiprocessors. *Information Processing Letters*, 84(2):93–98, 2002.

[48] The Apache Software Foundation. GridMix, 2015. `http://hadoop.apache.org/docs/current/hadoop-gridmix/GridMix.html`.

[49] B. Trushkowsky, P. Bodík, A. Fox, M. J. Franklin, M. I. Jordan, and D. A. Patterson. The scads director: Scaling a distributed storage system under stringent performance requirements. In *Proceedings of the 9th USENIX Conference on File and Stroage Technologies*, FAST'11, pages 12–12, Berkeley, CA, USA, 2011. USENIX Association.

[50] A. Tumanov, T. Zhu, J. W. Park, M. A. Kozuch, M. Harchol-Balter, and G. R. Ganger. TetriSched: Global rescheduling with adaptive plan-ahead in dynamic heterogeneous clusters. In *Proceedings of the Eleventh European Conference on Computer Systems*, EuroSys'16, pages 35:1–35:16, New York, NY, USA, 2016. ACM.

[51] V. K. Vavilapalli, A. C. Murthy, C. Douglas, S. Agarwal, M. Konar, R. Evans, T. Graves, J. Lowe, H. Shah, S. Seth, et al. Apache Hadoop YARN: Yet Another Resource Negotiator. In *Proceedings of the 4th annual Symposium on Cloud Computing*, page 5. ACM, 2013.

[52] S. Venkataraman, Z. Yang, M. Franklin, B. Recht, and I. Stoica. Ernest: efficient performance prediction for large-scale advanced analytics. In *13th USENIX Symposium on Networked Systems Design and Implementation (NSDI 16)*, pages 363–378, 2016.

[53] S. Verboven, P. Hellinckx, F. Arickx, and J. Broeckhove. Runtime prediction based grid scheduling of parameter sweep jobs. In *Asia-Pacific Services Computing Conference, 2008. APSCC '08. IEEE*, pages 33–38, Dec 2008.

[54] A. Verma, L. Cherkasova, and R. H. Campbell. Aria: Automatic resource inference and allocation for mapreduce environments. In *Proceedings of the 8th ACM International Conference on Autonomic Computing*, ICAC '11, pages 235–244, New York, NY, USA, 2011. ACM.

[55] A. Verma, L. Pedrosa, M. Korupolu, D. Oppenheimer, E. Tune, and J. Wilkes. Large-scale cluster management at Google with Borg. In *Proceedings of the Tenth European Conference on Computer Systems*, page 18. ACM, 2015.

[56] K. Wang and M. M. H. Khan. Performance prediction for apache spark platform. In *High Performance Computing and Communications (HPCC), 2015 IEEE 7th International Symposium on Cyberspace Safety and Security (CSS), 2015 IEEE*

12th International Conferen on Embedded Software and Systems (ICESS), 2015 IEEE 17th International Conference on, pages 166–173. IEEE, 2015.

[57] P. Wieder, J. M. Butler, W. Theilmann, and R. Yahyapour. *Service level agreements for cloud computing*. Springer Science & Business Media, 2011.

[58] M. Zaharia, D. Borthakur, J. Sen Sarma, K. Elmeleegy, S. Shenker, and I. Stoica. Delay Scheduling: A Simple Technique for Achieving Locality and Fairness in Cluster Scheduling. In *Proceedings of the ACM European Conference on Computer Systems*, EuroSys, 2010.

[59] Z. Zhang, L. Cherkasova, A. Verma, and B. T. Loo. Automated profiling and resource management of pig programs for meeting service level objectives. In *Proceedings of the 9th International Conference on Autonomic Computing*, ICAC '12, pages 53–62, New York, NY, USA, 2012. ACM.

The SNOW Theorem and Latency-Optimal Read-Only Transactions

Haonan Lu*, Christopher Hodsdon*‡, Khiem Ngo*, Shuai Mu†, Wyatt Lloyd*
**University of Southern California,* †*New York University*

Abstract

Scalable storage systems where data is sharded across many machines are now the norm for Web services as their data has grown beyond what a single machine can handle. Consistently reading data across different shards requires transactional isolation for the reads. Yet a Web service may read from its data store hundreds or thousands of times for a single page load and must minimize read latency to keep response times low. Examining the read-only transaction algorithms for many recent academic and industrial scalable storage systems suggests there is a tradeoff between their power—expressed as the consistency they provide and their compatibility with other types of transactions—and their latency.

We show that this tradeoff is fundamental by proving the SNOW Theorem, an impossibility result that states that no read-only transaction algorithm can provide both the lowest latency and the highest power. We then use the tight boundary from the theorem to guide the design of new read-only transaction algorithms for two scalable storage systems, COPS and Rococo. We implement our new algorithms and then evaluate them to demonstrate they provide lower latency for read-only transactions and to understand their impact on overall throughput.

1 Introduction

Scalable data stores are a fundamental building block of large-scale systems, such as modern Web services. Spreading data across machines—i.e., sharding—allows the system to scale its capacity and throughput, but also complicates how programs and users interact with the data. When all the data is on a single machine, consistently updating that machine is sufficient to ensure reads are consistent. When data is spread across machines, consistently updating the data store is no longer sufficient because reads to different shards will arrive at different times and thus see different views of the data store.

[‡]Work partially done as a student at Rutgers University-Camden.

Consistently viewing data thus requires transactional *isolation*, where reads to different shards either all observe a given update or none do. General transactions provide isolation, but are heavyweight and complex, especially for transactions that do not update data. Thus, it is common to have a special algorithm for *read-only transactions*, which are transactions the system knows will only read data. The importance of these read-only transaction algorithms has been recognized by many recent systems [5, 8, 11, 12, 14, 26, 27, 29, 31, 37, 38].

Read-only transaction algorithms ensure isolation, but often incur overhead relative to simple inconsistent reads of the same data. This overhead stems from extra rounds of communication to find a consistent view, extra metadata to determine if a view is consistent, and/or blocking operations until a consistent view is found. The overhead of these algorithms is important because many real-world workloads are dominated by reads and thus read performance determines the performance of the overall system. For instance, 99.8% of the operations for Facebook's distributed data store TAO are reads [10]. The latency of these reads is especially important because, as Facebook has reported, "a single user request may result in thousands of subqueries, with a critical path that is dozens of subqueries long" [4].

This breadth of reads and especially the depth of sequential reads for a single page load make their latency critical to the response times of the Web services. These response times are aggressively optimized because they affect user engagement and revenue [13, 24, 34]. Thus, one way to improve upon existing scalable storage systems is to decrease the latency of their read-only transactions. But instead of simply making them faster, we seek to make them *latency-optimal*, i.e., as fast as possible.

When examining existing systems we were able to derive latency-optimal read-only transaction algorithms for some, but not all of them. Investigating the cause of this dichotomy led us to discover a tradeoff between the latency and the power of read-only transactions.

We prove this tradeoff is fundamental with the SNOW

Theorem, which states it is impossible for a read-only transaction algorithm to provide all four desirable properties: **S**trict serializability, **N**on-blocking operations, **O**ne-response from each shard, and compatibility with conflicting **W**rite transactions. The power-related properties are strict serializability—which is the strongest form of consistency—and compatibility with conflicting write transactions—which indicates what other types of transactions are in the system. The latency-related properties are non-blocking operations—which ensures each shard immediately handles each read request—and one response—which ensures a single round of messages with the minimal amount of data.

The intuition of the proof is that when a transaction with writes commits there is a point at every server when the transaction becomes visible, but that the asynchronous nature of the network allows read requests to arrive before the transition on one server and after the transition on another. To cope with this possibility, a read-only transaction algorithm must either settle for consistency weaker than strict serializability (S), block some read requests to avoid the inconsistent interleaving (N), coordinate and/or retry the reads (O), or preclude the possibility of conflicting write transactions (W).

The SNOW Theorem is similar to the CAP Theorem [9, 18] in that it helps system designers avoid trying to achieve the impossible and identifies a fundamental choice they must make when designing their system. In addition, we make the SNOW Theorem even more useful by demonstrating what is possible. We show the four properties are tight by describing algorithms that provide each combination of them. We further tighten this boundary by moving beyond considering the properties as binary to instead viewing them as spectrums. For instance, we show that while strict serializability is impossible with the other three properties, an only-slightly weaker consistency model we call process-ordered serializability is possible. We call algorithms that touch the boundary of what is possible *SNOW-optimal*.

Using the lens of SNOW-optimality we can examine scalable data stores to determine if and what room for improvement in their read-only transactions exists. We find room for improvement in many systems, and focus in particular on two recent and quite different data stores, COPS [26] and Rococo [29]. COPS is scalable, geo-replicated, causally consistent, and has only read-only transactions and single-key write operations. In contrast, Rococo is scalable, designed for a single datacenter, strictly serializable, and has general transactions.

We present the design, implementation, and evaluation of novel read-only transaction algorithms for COPS and Rococo. We call the resulting systems COPS-SNOW and Rococo-SNOW. The key insight common to the systems is that to make reads as fast as possible we need to shift as much coordination overhead as possible into writes.

Our evaluation of COPS-SNOW shows that it almost always provides lower latency for read-only transactions and improves latency more as contention increases at the cost of lower overall throughput. Our evaluation of Rococo-SNOW shows that it always achieves lower latency for read-only transactions and has much higher throughput in the high-contention online transaction processing workloads Rococo is designed for, at the cost of slightly lower throughput under low contention.

The contributions of this paper include:

- The SNOW Theorem, which proves there is a fundamental tradeoff between the power and latency of read-only transaction algorithms. This paper also contributes algorithms that show the tightness of the SNOW Theorem and the precise boundary of what is possible, which we characterize as SNOW-optimality.
- The design and implementation of novel read-only transaction algorithms for both the COPS and Rococo scalable data stores that are latency-optimal and SNOW-optimal, respectively.
- Evaluations of COPS-SNOW and Rococo-SNOW that explore their effect on the latency and throughput of the systems under a variety of settings.

Section 2 presents necessary background and Section 3 explains the SNOW properties. Section 4 gives the statement and proof of the SNOW Theorem, shows its tightness, and explores SNOW-optimality. The designs of COPS-SNOW and Rococo-SNOW are presented in Section 5 and then evaluated in Section 6. Section 7 discusses related work and Section 8 concludes.

2 Background

Web services are typically built using two distinct tiers of machines: a frontend tier and a storage tier. The frontend tier is stateless and handles requests from users by executing application code that reads and writes data from the stateful storage tier. The Web service is typically replicated across multiple datacenters, but we restrict our discussion here to a single datacenter for simplicity.[1] The storage tier shards its data across many machines.

Figure 1 shows how a simple page is generated in a Web service. A frontend machine receives a request from a user and then runs the application logic to generate her page by reading data across many shards in the storage tier. All of the reads must complete before the page can be returned to the user. A typical page load issues hundreds or thousands of reads [4]. Many of these reads can be issued in a parallel batch like the reads of *a* and *b*. However, some reads are dependent on earlier reads, i.e.,

[1] Our results are magnified in cross datacenter settings.

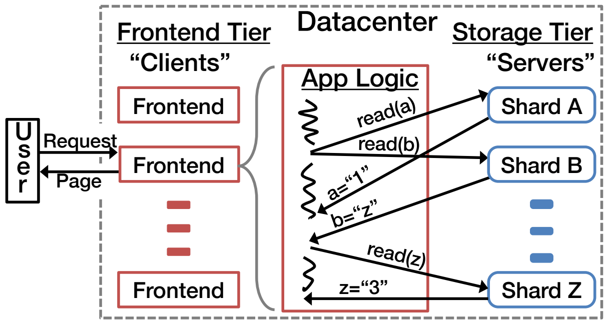


Figure 1: Typical Web service architecture with a frontend machine executing application logic to generate a page that reads data from the storage tier.

they read from keys that are returned by earlier reads. In Figure 1 the read of z depends on the read of b that returns z. Within a page load there are often chains of key dependences that are dozens of reads deep [4].

Because of the breadth and especially depth of these reads, providing low read latency is essential to enabling fast page load times. In the rest of this paper we focus on *one-shot* [20] read-only transactions that do not include key dependences that cross shards. One-shot read-only transactions can be issued in a single parallel batch, e.g., the reads of a and b. Following key dependences requires *multi-shot* read-only transactions. We focus on one-shot transactions because they are simpler to reason about and their results generalize: what is not possible for one-shot read-only transactions is also not possible for the more general multi-shot read-only transactions.

In this paper we consider read-only transactions generally, but with the current motivation of application logic on frontend machines consistently reading data from the storage tier to generate pages. To match general terminology on read-only transactions we call the frontend machines the *clients* and the storage tier the *servers*.

3 The SNOW Properties

This section introduces the SNOW properties and explains their importance for read-only transactions.

3.1 Strict Serializability

Strict serializability ensures there exists a total order over all the transactions in the system—i.e., transactions are *serializable*—and their results appear to have come from a single machine processing them one at a time [32]. This latter requirement ensures the total order respects the *real-time ordering* of transactions [19]. That is, if transaction t_2 begins in real time after transaction t_1 has completed, then t_2 will appear after t_1 in the total order.

When two transactions are concurrent there is no real-time ordering between them. For instance, if t_4 begins after t_3 begins but before t_3 has finished, then they are concurrent and either could be ordered first in a legal total order. The total order requirement of strict serializability guarantees that transactions are fully isolated, i.e., a transaction does not observe partial effects of other transactions. Informally, the real-time ordering requirement of strict serializability guarantees a read-only transaction always returns the most recent values.

Strict serializability is the most desirable consistency model because it provides the strongest guarantees. It is easiest for programmers to write correct application logic on top of a strictly serializable system, and it eliminates the most user-visible anomalies [28] compared to other consistency models. Section 4.4 discusses weaker consistency models.

3.2 Non-Blocking Operations

We define *non-blocking operations* to require that each server can handle the operations within a read-only transaction without blocking for any external event. That is, a process involved in handling a read-only transaction never voluntarily relinquishes a processor. Blocking behaviors that are prohibited include waiting for a lock to be available, waiting for messages from other servers, waiting for messages from other clients, or waiting for a timeout to fire. In contrast, non-blocking behavior ensures a server can immediately process and respond to requests from clients. Non-blocking operations are desirable because they directly relate to the latency of the read-only transactions; they save at least the time that would be spent blocking.

3.3 One Response Per Read

We define *one response* per read to be the combination of one round-trip to each server and one version per read. The *one version* subproperty requires that servers send only one value for each read. The *one round-trip* subproperty requires the client to send at most one request to each server and the server to send at most one response back. (This allows for zero messages to and from some servers, for instance, if they do not store data being read.)

The one version subproperty aligns with the latency of read-only transactions. If a server sends multiple versions of a value, that much more time is spent serializing, transmitting, and deserializing the values. The one round-trip subproperty strongly aligns with the latency of read-only transactions. For instance, an algorithm that takes two round trips will take roughly twice as long in transmission and queuing. This subproperty also disallows algorithms that abort a transaction and then start over, because starting over is another round trip. We explore multi-round algorithms further in Section 4.4.

The one response property is desirable because it leads to faster read-only transactions. We call algorithms that provide the one response and non-blocking properties *latency-optimal* because they are as fast as possible: a client sends a single request to each server, each server handles the request immediately, and the servers send back exactly the data the client wants to read.

3.4 Write Transactions that Conflict

We define the *write transactions* property as the ability of a read-only transaction algorithm to coexist with conflicting transactions that update data. This requires, first, that a data store allows transactions that update data. General transactions that read and write data satisfy this requirement, as do weaker write-only transactions that only write data. This property also requires that write transactions can *conflict* with read-only transactions, i.e., write transactions can update data spread across multiple servers concurrently with read-only transactions viewing that data. The ability to coexist with conflicting write transactions is desirable because write transactions make programming application logic much easier.

4 The SNOW Theorem

The SNOW Theorem is an impossibility result that states no read-only transaction algorithm can provide all of the SNOW properties. This section presents a proof of the SNOW Theorem, discusses the tightness of the theorem, defines SNOW-optimality, and discusses the spectrums of related properties.

4.1 Models, Definitions, and Assumptions

System Model. Our system model is similar to that used in FLP [16]. A distributed system is modeled by a set of N processes, where $N > 1$. Processes communicate by sending and receiving messages. A set of client processes (machines) issue requests to the server processes (machines), which store the data. System actions are modeled as each process going through a sequence of events, where an event is an atomic step of receiving a message, doing local computation, and/or producing a set of output messages.

Network Model. The SNOW Theorem holds for the asynchronous network [18] and the partially synchronous network [15] models. In an asynchronous network, there are no physical clocks and messages between processes can be arbitrarily delayed. In a partially synchronous network, the message delay is bounded and there is a bound on the drift rate between clocks at different processors, but either the rates are not known apriori or do not hold immediately. In the proof, we use an asynchronous network for simplicity. We then discuss the correctness of the SNOW Theorem under the partially synchronous network model.

Definitions. A transaction is a set of operations that read and/or update data. Clients group all operations that are sent to the same server into a single request. The *invocation time* of the transaction is the time when a client process sends each request in the transaction to the involved servers. The *response time* of the transaction is the time when the client has received all the responses from the servers.

Lamport's *happened-before* relation [21] is the (smallest) partial order such that "1) If a and b are events in the same process, and a comes before b, then $a \rightarrow b$. 2) If a is the sending of a message by one process and b is the receipt of the same message by another process, then $a \rightarrow b$. 3) If $a \rightarrow b$ and $b \rightarrow c$ then $a \rightarrow c$."

We use the happened before relation to differentiate between two server behaviors for handling read-only transaction requests. Let r be the handling of a request from a read-only transaction, R, on a server. A write transaction, W, is *unknown* to r if the client that issued R could not know about W when it issued the request and the server handling the request could not know about R until the request arrived. More formally, W is unknown to r if $W_{inv} \nrightarrow R_{inv}$ and $R_{inv} \nrightarrow e'$, where e' is the event on server S that directly precedes r. The server handles r with the *default* behavior if W is unknown to r.

Assumptions. We assume a reliable network, reliable processors, and one-shot transactions. A reliable network eventually delivers every message sent. Reliable processors eventually receive and process every message sent. One-shot transactions [20] require at most one request per server process, i.e., they do not use the output of a request as part of the input of another request.[2] These assumptions are not necessary as the SNOW Theorem holds without them. Assuming them demonstrates the strength of the impossibility result. We also use these assumptions when characterizing what is possible (§4.3–4.4) and categorizing related work (§7).

We also assume there are at least two server processes and at least three client processes. These assumptions are necessary for our proof. SNOW is possible with a single server process or a single client process. It is an open question if SNOW is possible when the system has at least two server processes and exactly two client processes.

[2]Equivalently, there are no cross-server key- or value-dependencies.

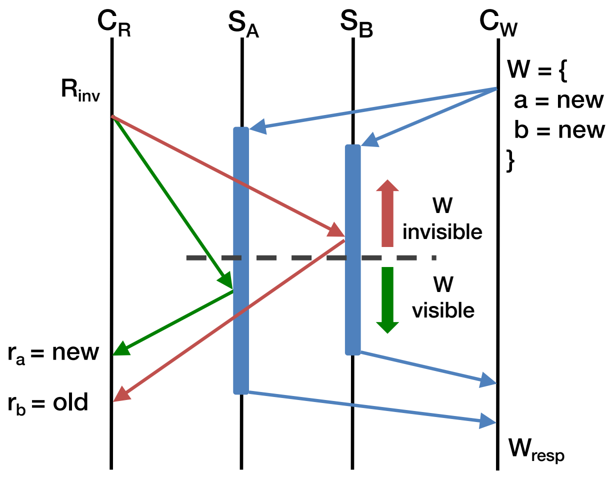


Figure 2: The asynchronous nature of the network allows one request in a read-only transaction to arrive before a conflicting write transaction is visible on one server, while another request arrives after the write transaction becomes visible at a different server. This requires read-only transactions to either settle for weaker consistency (S), block some read requests (N), coordinate and/or retry the reads (O), or preclude conflicting write transactions (W).

4.2 SNOW is Impossible

The SNOW Theorem is an impossibility result that states no read-only transaction algorithm can provide optimal latency and the highest power. Providing optimal latency requires providing the non-blocking (N) and one response (O) properties. Providing the highest power requires providing strict serializability (S) and being compatible with conflicting write transactions (W).

The intuition of the proof is that when a transaction with writes commits, there is a point at every server when the transaction becomes visible, i.e., newly arriving reads will see its effect. However, the asynchronous nature of the network allows read requests to arrive before the transition on one server and after the transition on another, as shown in Figure 2. Turning this intuition into a proof, however, turns out to be more complex. We prove the SNOW Theorem by contradiction, i.e., we assume there exists a read-only transaction with all of the SNOW properties and then eventually show this assumption leads to a contradiction. First, we show that the default behavior of servers must be to return a value, which we called the "default" value (Lemma 1). Second, we show that this default value must initially not expose an ongoing write transaction (Lemma 2). Third, we show that this default value must eventually expose that write transaction (Lemma 3). Fourth, we show that this default value must transition from not exposing the write transaction to exposing it, at some point on each server (Corollary 4). Finally, we prove the SNOW Theorem by constructing an execution where two requests from a read-only transaction take the default behavior at two different servers, with one request arriving before the transition and one arriving after the transition.

The SNOW Theorem. *No read-only transaction algorithm provides all of the SNOW properties.*

Proof. Assume to contradict that there exists a read-only transaction algorithm with all of the SNOW properties. Consider a distributed system with at least the two servers, S_A and S_B, and the three clients, C_R, $C_{R'}$, and C_W, that we assumed exist.

Let R be a read-only transaction issued by C_R that reads a from S_A and b from S_B. Let the first events that happen on S_A and S_B as part of the read-only transaction algorithm be r_a and r_b, respectively. Let W be a conflicting write transaction that writes $a = new$ and $b = new$. Let the values of a and b before W is applied be old.

Lemma 1. *The default behavior at servers returns values that are used by clients.*

Proof. This follows directly from the non-blocking and one response properties. ■

Lemma 2. *Servers initially return old by default.*

Proof. R and W can be concurrent by the conflicting write transaction property. r_a can occur at S_a before any event that happened after W_{inv} by the concurrency of R and W and the asynchronous network. r_a can be the first request in R to arrive at a server by the asynchronous network. Then, by definition, r_a is handled by default and returns a value by Lemma 1. S_a cannot know of W because $W_{inv} \nrightarrow r_a$ and so must return old. ■

Lemma 3. *Servers eventually return new by default.*

Proof. Assume to contradict that servers never return *new* by default. Let R be invoked after W returns at C_W. R must return $a = new$ and $b = new$ by strict serializability. By assumption, S_a and S_b must have returned *new* by a non-default behavior. Let r_a be the first of the requests of R to be handled by a server, by the asynchronous network. Then the non-default behavior must have been triggered by the receipt at C_R of some message m_{see} that connects $W_{inv} \rightarrow R_{inv}$.

Consider the execution up until the point where the m_{see} message is in the network. By the asynchronous network we can deliver or delay messages arbitrarily. Delay all messages not explicitly mentioned and continue that execution by delivering m_{see}.[3] Then deliver all messages for R, which must still see $a = new$ and $b = new$

[3]This construction guards against write transaction algorithms that explicitly notify all clients of their existence before completing.

because they are indistinguishable from the original execution. Let R' be a read-only transaction issued by $C_{R'}$ that reads a from S_A and b from S_B that is invoked after R returns. Deliver no messages to $C_{R'}$ before it issues R', and deliver all of the messages in R'. Let r'_a be the first request in R' delivered by the asynchronous network. By definition r'_a is handled by default, and by assumption *old* must be returned. By strict serializability, *old* must also be returned for b to R'. Thus, R' reads $a = old$ and $b = old$ even though it is invoked after R returns having read $a = new$ and $b = new$. This violates strict serializability and is our contradiction. ■

Corollary 4. *There exists a transition at each server between defaulting to old and defaulting to new.*

Proof. This follows directly from Lemmas 2 and 3. ■

Proof of the SNOW Theorem. Let t_A and t_B be the first transitions from defaulting to old and defaulting to new at S_A and S_B, respectively, that exist by Corollary 4. Let t_A happen first, without loss of generality. Deliver no message to C_R before R is invoked. Deliver r_a immediately after t_A and before r_b. By definition, r_a is handled by default, and by being immediately after t_A it must return *new*. Next, immediately deliver r_b. By definition, r_b is handled by default. By being before t_B, r_b must return *old*. Thus, in this execution R returns $a = new$ and $b = old$. This violates strict serializability and is the contradiction we needed to prove the SNOW Theorem. ■

SNOW with Partial Synchrony. We have shown the correctness of the SNOW Theorem for the asynchronous network model. The theorem also holds for partially synchronous networks because transforming bounds on delay or clock drift into knowledge that can be used in an algorithm requires blocking in proportion to those bounds [15]. Lemma 2 still holds because r_a may still arrive before knowledge of W and would need to wait for that knowledge to arrive to be able to return *new*, which is not allowed because it is blocking. Lemma 3 also still holds because eliminating the possibility of C_R receiving m_{see}, then completing a read-only transaction, and then $C_{R'}$ completing a read-only transaction before receiving any messages would require waiting out bounds. That waiting would have to be on the read path of either C_R and/or $C_{R'}$, which is not allowed because it is blocking.

4.3 Tightness and SNOW-Optimality

We demonstrate the tightness of the SNOW Theorem by showing that every combination of three out of the four SNOW properties is possible. That is, there exists read-only transaction algorithms that satisfy S+O+W, N+O+W, S+N+W, and S+N+O.

Rococo-SNOW, one of the algorithms we will discuss later in this paper, is S+O+W. It is a blocking algorithm but is compatible with stronger transaction semantics, i.e., write-only transactions and general transactions. Algorithms that provide multi-object snapshots in the past are often N+O+W algorithms. For instance, Spanner [11]'s snapshot reads API that is serializable. MySQL Cluster [31] also uses a N+O+W algorithm while providing read committed consistency. These algorithms favor low latency over the isolation and/or recency of strict serializability. Eiger [27]'s read-only transaction algorithm is S+N+W, as it uses multiple rounds to make the write transaction known to all reads.

We designed a novel algorithm for COPS, COPS-DW, that satisfies S+N+O by making all simple write operations go through a distinguished writer. The distinguished writer totally orders the writes. In addition, when each write commits, the distinguished writer computes a consistent snapshot for concurrent or later read-only transactions to return. This algorithm is of theoretical interest only and is not practical because it serializes all writes. It remains open whether there exists practical S+N+O algorithms. These algorithms demonstrate the tightness of the SNOW Theorem, that is, the four SNOW properties are the minimal set of properties that make coexistence impossible.

Given that the SNOW Theorem is tight, we define a read-only transaction algorithm to be *SNOW-optimal* if its properties sit on the boundary of the SNOW Theorem, i.e., it achieves three out of the four SNOW properties. In the four different combinations of SNOW-optimality, S+N+O and N+O+W favor the performance of read-only transactions since non-blocking and one response lead to low latency. We call algorithms that satisfy N+O *latency-optimal*. In contrast, property combinations S+O+W and S+N+W lean towards the power of read-only transactions as they provide the strongest consistency guarantee and compatibility with conflicting write transactions.

4.4 Spectrums of Properties

To fully understand what we can learn from the SNOW Theorem, we further tighten the boundary on what is possible by moving beyond considering the one response and strict serializability properties as binary to viewing them as spectrums.

If the one response property is sacrificed, then how many rounds of messages are sufficient for the rest of the properties to hold?

By examining the systems we have found, existing read-only transaction algorithms range from at most three rounds of messages (Eiger) to an unbounded number of rounds of messages. It is currently open if there exist S+N+W algorithms with at most two rounds.

If the strict serializability property is sacrificed, then what is the next strongest consistency model an algorithm can achieve if the other three properties hold?

Process-ordered serializability is a consistency model slightly weaker than strict serializability that is effectively the combination of serializability and sequential consistency [22]. It requires that there exists a legal total ordering over all operations in the system, provides transactional isolation, and guarantees that the legal total order agrees with each process's ordering of its own operations. It is weaker than strict serializability in that it does not necessarily return the most recent values across processes. We designed a novel algorithm we call Eiger-PS for the Eiger data store. Eiger-PS satisfies N+O+W, while providing process-ordered serializability.

The key idea of the read-only transaction algorithm of Eiger-PS is for each client to maintain a serializable view of the system and to only move to a new view when it is certain that the new view contains that client's most recent write. We accomplish this by having each client maintain a *global safe time*, GST, which is the latest logical time at which no server is holding a pending write transaction. The GST is maintained by each client periodically requesting from every server their *local safe time*, which is the latest time on the server with no in-progress write transactions. The client then only reads at its GST, which provides serializability in Eiger [27]. To achieve process-ordered serializability, each client must see its most recent write. Reading at the GST will not necessarily guarantee that the client sees its most recent write as the client may not have updated its GST since the commit of the last write transaction. Thus, we have each write transaction wait to return until the writing client's GST exceeds the logical commit time of the write transaction. This does not provide strict serializability because a client may not see the most recent write committed by another client.

Because process-ordered serializability is possible, all of the consistency models that are weaker than process-ordered serializability are also able to coexist with the other three properties. For instance, some weaker consistency models include serializability, causal consistency, snapshot isolation, parallel snapshot isolation, and read committed. That is, with all of these weaker forms of consistency, a read-only transaction algorithm can provide N+O+W.

5 Read-Only Transaction Designs

This section explores how to use SNOW-optimality as a lens to examine existing algorithms, our common insight in deriving SNOW-optimal algorithms, and the designs of COPS-SNOW and Rococo-SNOW that integrate new read-only transaction algorithms.

5.1 Exploring Improvements with SNOW

SNOW-optimality is a powerful lens with which to examine the design of read-only transaction algorithms. If an algorithm is already SNOW-optimal, then we cannot improve it without making a different choice in the trade-off between latency and power. If an algorithm is not SNOW-optimal, however, we know that it is possible to improve it without making a different design choice.

Any algorithm that is not SNOW-optimal has at most two of the SNOW properties. We improve upon such algorithms by keeping the SNOW properties they provide and adding at least one of the latency-related properties. We do not add strict serializability or compatibility with conflicting write transactions because doing so would change the base system into something new.

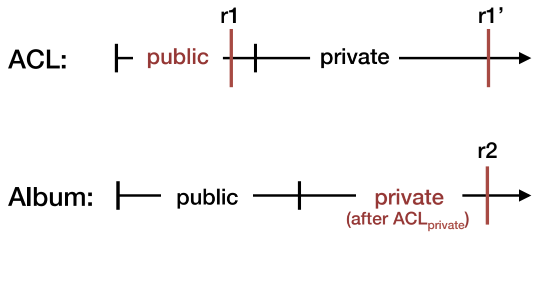
The COPS distributed data store has a non-blocking algorithm for its read-only transactions and we designed a new non-blocking and one response algorithm. The new COPS-SNOW algorithm is latency-optimal, but not SNOW-optimal because it is neither strictly serializable nor compatible with write transactions. The Rococo distributed data store has a read-only transaction algorithm that is strictly serializable and compatible with conflicting writes. We designed a new algorithm that adds the one response property. The new Rococo-SNOW algorithm is SNOW-optimal but not latency-optimal.

In addition to helping us discover systems whose algorithms we can improve, the SNOW Theorem also helps us avoid trying to improve systems that we cannot. Some of the distributed data stores we examined were already SNOW-optimal and so we knew it would be impossible to improve them. For instance, Spanner [11] is SNOW-optimal because it has a strictly serializable, one round read-only transaction algorithm that is compatible with conflicting write transactions. Section 7 discusses more systems that are already SNOW-optimal.

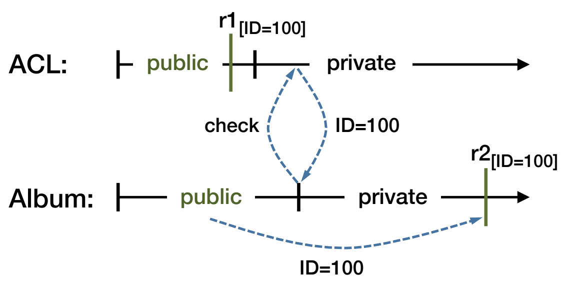
5.2 Common Insight for Optimal Reads

After identifying if and how we can improve upon the read-only transaction algorithm in existing systems, we need to design algorithms that realize that improvement. We have found one common insight in our new algorithms that we think will be useful in deriving other SNOW-optimal algorithms. This key insight is to make reads cheaper by making writes more expensive.

Instead of blocking reads, block writes. Instead of requiring extra rounds of communication for reads, require them for writes. Shifting the burden to writes will always improve the individual performance of reads. But for the read-heavy workloads that are common for Web services, such a design can also improve overall performance because it diminishes a minority of the workload to improve the majority of it.



(a) COPS read-only transaction



(b) COPS-SNOW read-only transaction

Figure 3: Ensuring causal consistency for a private photo that is updated after setting the ACL to private. COPS takes two rounds of requests (r_1,r_2 then $r_{1'}$) while COPS-SNOW takes only one round to ensure consistency.

5.3 COPS-SNOW Design

This subsection describes the COPS system and our new read-only transaction algorithm for it.

COPS Overview. COPS is a scalable, geo-replicated storage system where each replica (datacenter) contains a full copy of the data sharded across many machines. COPS has single-key write operations and does not have write transactions. Each datacenter accepts writes locally and then replicates them to remote datacenters with metadata that indicates their causal dependencies. Remote datacenters check that the write's causal dependencies are satisfied before applying the write. This ensures that the data spread across many shards in each replica is always causally consistent [3, 21]. COPS has read-only transactions that are handled entirely locally in a replica and provide a causally consistent view of the data store. For the rest of our discussion of COPS we focus on its operations within a single datacenter, but our results apply equally in the geo-replicated setting.

The original read-only transaction algorithm in COPS is causally consistent, non-blocking, two rounds, and is not compatible with write transactions. COPS read-only transactions begin when application logic invokes a read-only transaction that includes the full list of keys the client wants to read. The client then sends out a first round of read requests to each shard that has data in the transaction. Servers respond with their current value for the data along with the causal dependencies of each value. After the first round the client checks to see if all of the returned values are mutually consistent. Each causal dependency that is returned with a read is a constraint on other values in the system, e.g., b_1 depends on a_1 means that if a client observes b_1 it must also observe a_1 or an even later version of a (if it reads a). If all of the dependencies of all the values returned in the first round are satisfied, then COPS returns the values to the application logic after a single round. If, however, not all of the dependencies are satisfied, then COPS issues a second round of read requests for each value that does not satisfy other values' dependencies. COPS requests the specific versions of keys that are depended upon, e.g., if a_0 and b_1 are returned in the first round then a_1 will be requested in the second round. Requesting these specific versions guarantees COPS will complete in two rounds because these versions satisfy current dependencies. In addition, these specific versions do not introduce any new dependencies because, by the definition of causality, their dependencies are a subset of the dependencies of values that depend upon them, e.g., a_1's dependencies are a subset of b_1's.

COPS-SNOW Algorithm. We improve COPS with a new latency-optimal read-only transaction algorithm. Our COPS-SNOW algorithm keeps the power properties of the current algorithm: it provides causal consistency and is not compatible with conflicting write transactions. It is latency-optimal because it keeps the non-blocking property of the current algorithm and adds the one response property. Following our common insight we shift the complexity from the reads to the writes in COPS. More specifically, we shift the consistency check and second round fetch of consistent values from the read-only transaction algorithm into the write algorithm.

In COPS a second-round read is needed if and only if one part of the read-only transaction r_a does not see a write w_{a1} and another part of the read-only transaction r_b does see a write w_{b1} that is causally after w_{a1}. Figure 3a shows this in action with COPS using the canonical access control list (ACL) and photo example where an album is switched to private (w_{a1}) and then a private photo is added (w_{b1}). In this example, COPS will send a new read $r_{1'}$ that will see write w_{a1} and return the consistent set of the private ACL and Album.

Our new algorithm flips this responsibility by having a write check if any of its causal dependencies have not been observed by an ongoing read-only transaction. If

```
Client Side
function read_only_txn(<keys>):
  trans_id = generate_uuid()
  vals, deps = []
  for k in keys # in parallel
    vals[k], deps = read_txn(k, trans_id)
  # update causal dependencies
  return vals

function write(key, val):
  old_deps = get_deps()
  new_dep = write(key, val, old_deps)
  # update causal dependencies
  return

Server Side
function read_txn(key, trans_id):
  if trans_id in old_rdrs[key]
    time = old_rdrs[key][trans_id]
    return val = read_at_time(key, time)
  curr_rdrs[key].append(trans_id,
                       logical_time.now())
  return read(key)

function write(key, val, deps):
  for d in deps # in parallel
    old_rs = check_dep(d)
    old_rdrs[key].append(old_rs)
  old_rdrs[key].append(curr_rdrs[key])
  curr_rdrs[key].clear()
  write(key, val)
  # calculate causal dep for this write
  return new_dep

function check_dep(dep)
  # normal causal dependency check
  return old_rdrs[dep.key]
```

Figure 4: Pseudocode for COPS-SNOW.

any of those causal dependencies were not observed by a read-only transaction then this write should not be observed by it either, so the write updates metadata encoding that. Figure 3b shows our new algorithm in action on the same example. COPS-SNOW returns the consistent set of the public ACL and Album.

Figure 4 shows the pseudocode for COPS-SNOW. On the client side writes are the same as in COPS, and read-only transactions are similar but simpler because they return the values from the first and only round. On the server side there are five high-level changes relative to COPS: two changes to reads, two to writes, and one to dependency checks.

The first change to reads is that they check to see if their enclosing transactions are listed in the old readers data structure (`old_rdrs`) and if so return an older, consistent value. The second change to reads is that they are recorded as observing the current value in the current readers data structure (`curr_rdrs`). This enables writes that overwrite the value to record which read-only transactions did not see them.

The first change to writes is that they do dependency checks to see if any of their causal dependencies overwrote values that a read-only transaction observed. If so, the write records in the old readers data structure that those read-only transactions should see older values to be consistent. The second change to writes is that they record any read-only transactions that observed the value they overwrote by copying the current readers data structure into the old readers data structure. The change to dependency checks is that they return the set of read-only transactions that did not see a causally dependent update. The combination of changes to writes enables this. Adding reads of the overwritten values captures reads that did not observe this write. Adding reads that did not see this write's causal dependencies—which also did dependency checks that added their causal dependencies, and so on—captures the transitive closure of this write's dependencies.

For clarity the pseudocode excludes logic related to updating causal dependencies; grouping reads to keys that are stored on the same server; updating Lamport clocks; and storing, reading, and garbage-collecting old versions. All of this logic is similar to what COPS does, and is identical to what Eiger does.

5.4 Rococo-SNOW Design

This subsection describes the Rococo system and our new read-only transaction algorithm for it.

Rococo Overview. Rococo is a strictly serializable, distributed data store with general transactions [29]. Rococo was designed primarily for the single datacenter setting we consider here. Rococo introduced a new concurrent control algorithm that outperforms traditional concurrency control algorithms under high contention workloads by reordering conflicting transactions instead of aborting them. Rococo requires the transactions that it executes to be chopped into pieces and analyzed for safety before the system is deployed. Each transaction is chopped into pieces that execute on shards as stored procedures. For instance, to increment keys *a* and *b* that are stored in different shards, Rococo would have a piece for *a* that invokes the increment stored procedure server-side and a separate piece for *b* that invokes the increment stored procedure server-side. Rococo analyzes the pieces of transactions to ensure that if they conflict at run time it will be able to safely reorder them.

Rococo's general transactions operate in three phases run by a coordinator. The first phase distributes pieces of the transaction to the appropriate shards and determines all directly conflicting transactions. The second phase ensures all shards have the same metadata about directly conflicting transactions. The third phase, which can often be skipped, ensures all shards have the same metadata about transitively (but not directly) conflicting transactions. After the second or third phase each shard deterministically orders all conflicting transactions and then executes them in that order.

The original read-only transaction algorithm in Rococo is strictly serializable, blocking, multi-round, and compatible with conflicting write transactions. It takes two rounds in the best case and an infinite number of rounds in the worst case. In the first round the coordinator sends read requests to each involved shard. Those read requests block until after the execution of all conflicting transactions that started at that shard before this read arrived. The second round is identical, and Rococo considers the read-only transaction successful only if both rounds read the same values. If not, Rococo will continue issuing another round of reads until two consecutive rounds return the same results. This algorithm ensures strict serializability for the reads because it ensures they are totally ordered relative to all conflicting transactions. Waiting for all conflicting transactions to execute at a shard before returning in the first round ensures those transactions will have at least started at all other involved shards before the second-round read arrives. Thus, if a read-only transaction is not fully ordered before or after a write transaction, it will see different results and continue trying.

Rococo-SNOW Algorithm. We improve Rococo with a new SNOW-optimal read-only transaction algorithm. Our Rococo-SNOW algorithm keeps the power properties of the current algorithm: it provides strict serializability and is compatible with conflicting write transactions. It also adds the one response property to these, which makes it SNOW-optimal. It is not latency-optimal because it blocks, which we know is unavoidable. Following our common insight we shift the complexity from reads into the commit algorithm of Rococo.

Due to space limitations, we only briefly describe our new algorithm and omit its pseudocode. It is conceptually similar to the COPS-SNOW algorithm in that it tracks whenever a value is read by a read-only transaction and then propagates the knowledge of that to all other servers where that ordering is important. In Rococo the second round (and additional rounds after that) are necessary to protect against the case where one part of a read-only transaction does not see a write transaction but another part does. Our new algorithm ensures this case never occurs by blocking each piece of a read-only transaction until all conflicting write transactions have executed at that shard. Rococo's commit algorithm ensures that each piece of a transaction has knowledge of the transitive closure of all conflicting transactions. We piggyback the knowledge of read-only transaction pieces that did not see any of the transitive closure of conflicting transactions into that commit algorithm.

If a different piece of the read-only transaction did not see a conflicting write transaction, then this shard will know about that through the commit algorithm before it unblocks this piece of the read-only transaction. Thus, the shard will know whether to return an old state or the most recent state when it executes the read-only transaction piece. When the coordinator receives replies from all involved shards, it knows the results are consistent and thus returns them to the application logic.

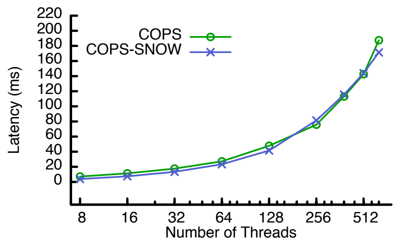
6 Evaluation

We experimentally evaluate COPS-SNOW and Rococo-SNOW to understand how their latency and throughput compare to the original COPS and Rococo under a variety of settings. The evaluation shows that both COPS-SNOW and Rococo-SNOW achieve lower latency for read-only transactions. COPS-SNOW achieves this at the cost of lower system throughput. Rococo-SNOW has slightly lower throughput than Rococo under low contention but actually achieves much higher throughput in the high-contention settings Rococo was designed for.

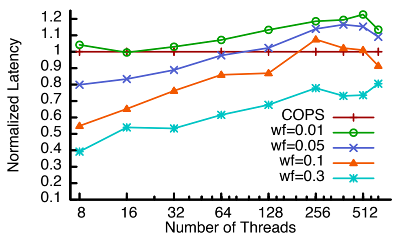
6.1 COPS-SNOW

Implementation. We implemented COPS-SNOW as a modification to Eiger [27], the successor to COPS. Eiger provides the same level of consistency guarantees as COPS, adds support for write-only transactions, supports a richer data model, and has a read-only transaction algorithm based on logical time instead of causal dependencies. We disable Eiger's write-only transactions to change it to a COPS-like mode where it has an at most two-round read-only transaction algorithm. This allows Eiger to propagate and store far fewer dependencies than COPS and makes its read-only transaction algorithm far more efficient. For this reason, we implement on top of Eiger: we are comparing to the state-of-the-art read-only transaction algorithm for causally-consistent systems without compatibility with write transactions. We refer to this baseline as COPS throughout the evaluation. This implementation is available publicly on GitHub.[4]

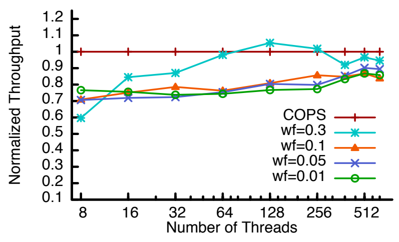
[4]`https://github.com/USC-NSL/COPS-SNOW`



(a) Latency with 0.1 write frac



(b) Normalized median latency



(c) Normalized throughput

Figure 5: Latency and throughput for COPS-SNOW and COPS for varying fractions of writes in the workload. The latency graphs show the median latency for read-only transactions. Throughput graphs show the overall throughput of the cluster.

Testbed, Bottleneck Resource, and Trials. We tried to match our experimental setup to that of Eiger's as much as possible. We ran all the experiments on the PRObE Nome testbed [17]. (Some of Eiger's experiments were run on Nome's predecessor Kodiak.) Each Nome machine has four Quad-Core AMD Opteron 2.2 GHz CPUs, 32 GB RAM, and two network interfaces: one 1 Gbps Ethernet and one 20 Gbps Infiniband. All COPS-SNOW experiments were run on a 20 Gbps Infiniband network, which matches the network configuration Eiger used. Due to inefficient thread scheduling within Cassandra, upon which Eiger is based, one instance cannot saturate all 16 cores on a physical machine. To make a fair baseline for comparison, we run two instances on each node so we can saturate one of the machine's resources. For all experiments the bottleneck is network interrupt processing. We ran 15 trials for each experiment and report the median. Each trial lasted at least 90 seconds with the first and last quarter excluded to avoid artifacts due to warm up, cool down, and imperfectly synchronized clients.

Configuration and Workloads. We evaluate COPS-SNOW using two logical datacenters that are physically co-located in the testbed with eight server machines each. We use 16 client machines to load the servers in one of the logical datacenters. (Our throughput disadvantage would decrease as client load shifted to be more evenly distributed across the datacenters.) We use the dynamic workload generator from Eiger with Zipfian traffic generation using these parameters:

Parameter	Default	Range
Value Size (B)	128	
Cols/Key	5	
Keys/Operation	5	5 – 32
Write Fraction	0.1	0.01 – 0.5
Zipfian Constant	0.8	0.7 – 0.99

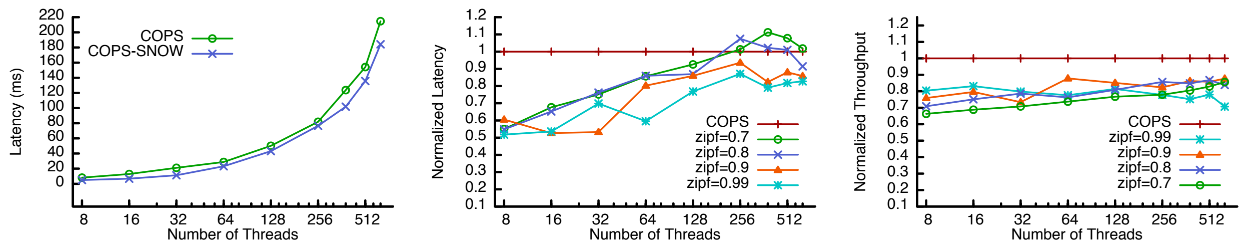
The default parameters match the defaults in Eiger's evaluation and we choose 0.8 as the default Zipfian constant because it provides moderate skew. For a write of 5 – 32 keys we send out 5 – 32 parallel, unrelated individual write operations. We explore how the throughput of COPS-SNOW compares to COPS under a variety of settings shown by the ranges of parameters we explore.

Performance with Varying Write Fraction. Figure 5 shows the latency and throughput of COPS-SNOW and COPS as we increase the number of closed-loop client threads on each client machine. Figure 5a shows the latency with a write fraction of 0.1. In this setting, there are enough writes to require COPS to sometimes go to the second round of its algorithm and COPS-SNOW has a slight latency advantage.

Figure 5b shows the median latency of COPS-SNOW normalized against that of COPS for a variety of write fractions. When the write fraction is very low at 0.01, there are so few writes that all of COPS's read-only transactions take only one round and have similar latency to COPS-SNOW. When the number of closed-loop client threads is high, sometimes the latency of COPS-SNOW is actually higher than that of COPS because the systems are overloaded. This overload is outside the model we considered in this paper, which we believe is reasonable because overload is outside of the normal range of system operations. Exploring the affect of overload on latency is an interesting avenue of future work.

The larger result from Figure 5b is that the latency improvement of COPS-SNOW increases as the write fraction increases. It is substantial when the write fraction is close to 0.3. The rest of the experiments use the default 0.1 fraction. If they used a higher write fraction their latency results would be more pronounced and if they used a smaller write fraction their latency results would be less pronounced.

Figure 5c shows the throughput of the cluster in the same settings. Here we see that COPS-SNOW is trading



(a) Latency with Zipf=0.9 **(b) Normalized median latency** **(c) Normalized throughput**

Figure 6: Latency and throughput for COPS-SNOW and COPS for varying levels of skew in the workload. Latency graphs show the median latency for read-only transactions. Throughput graphs show the overall throughput of the cluster.

away throughput for lower latency for read-only transactions. This loss in throughput comes from the extra messages in the write algorithm in COPS-SNOW.

Performance with Varying Keys per Operation. Due to space constraints, we omit the figure showing the throughput and latency of COPS-SNOW and COPS for a varying number of operations in each transaction. Our results show that COPS-SNOW has a latency advantage that increases as read-only transactions increase in size because the read-only transactions in COPS are more likely to go to the second round. It also shows that the throughput of COPS-SNOW becomes worse relative to COPS as read-only transactions become larger because each write has more causal dependencies to check.

Performance with Varying Levels of Skew. Figure 6 shows the latency and throughput of COPS-SNOW and COPS for varying levels of skew in the workload. Figure 6a shows the latency of read-only transactions when the skew is moderate with a Zipfian constant of 0.9. COPS-SNOW has a latency advantage over COPS because there is moderate contention in the workload and COPS sometimes needs a second round for reads. Figure 6b shows that COPS-SNOW has an increasing latency advantage as the workload becomes more skewed as long as the systems are not overloaded. Figure 6c shows that the throughput disadvantage of COPS-SNOW decreases slightly as the workload becomes more skewed and the extra RPCs in the write algorithm of COPS-SNOW are offset by the extra RPCs in the second round of read-only transaction in COPS.

6.2 Rococo-SNOW

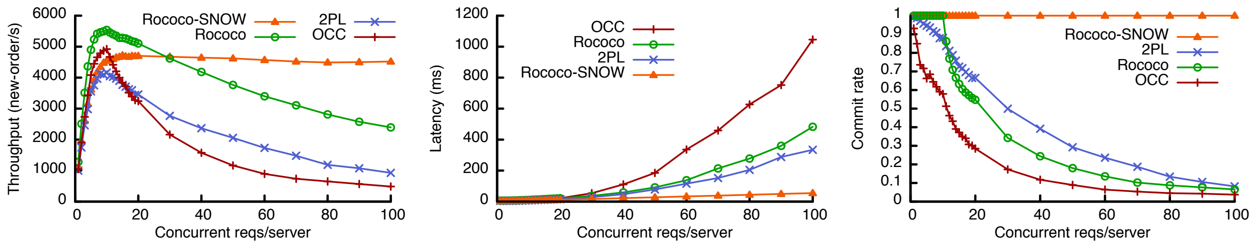
Implementation. We implemented Rococo-SNOW as a modification to Rococo's code base. We converted Rococo from a single-version to a multi-version system to support our read-only transactions and replaced its read-only transaction logic with our new algorithm. This implementation is available publicly on GitHub.[5]

Testbed, Bottleneck Resource, and Trials. We tried to match our experimental setup to that of Rococo's as much as possible. We ran all the experiment on the PRObE Nome testbed. (All of Rococo's experiments were run on Nome's predecessor Kodiak, which has been decommissioned.) The machines' specifications are the same as they were for COPS-SNOW with two exceptions. First, we use the Ethernet network interface instead of the Infiniband interface to match the setup from Rococo's evaluation (the network is never a bottleneck). Second, Rococo is single-threaded and used only one core in its evaluation so we run one Rococo process per machine, which only uses one of the cores. This core is always the bottleneck. The rest of our experiment settings were identical to what were used in Rococo, e.g., each trial lasted at least 60 seconds with first and last quarter excluded to avoid artifacts due to warm up, cool down, and imperfectly synchronized clients.

Configuration and Workload. We evaluate Rococo-SNOW using 8 server machines and 16 client machines. We evaluate using Rococo's district-sharded TPC-C with all parameters matching Rococo's evaluation [29].

TPC-C Throughput and Read-only Transactions. Figure 7 shows the performance of Rococo-SNOW, Rococo, 2PL, and OCC as load and contention are increasing by increasing the number of concurrent requests per server in the system. The throughput for the (read-write) new order transaction is shown in Figure 7a. The TPC-C benchmark requires a specific mix of its five transaction types, so this throughput is proportional to the throughput of each type of transaction. Our results for Rococo,

[5]`https://github.com/USC-NSL/Rococo-SNOW`



(a) New order throughput **(b) Stock level latency** **(c) Stock level commit rate**

Figure 7: Rococo-SNOW performance with Rococo's TPC-C benchmark. The throughput of new order transactions is shown, which is proportional to the throughput of all transactions. The median latency and commit rate of the read-only stock level transactions are shown.

2PL, and OCC, match what was observed in Rococo's evaluation. We see that Rococo-SNOW provides lower peak throughput than Rococo. This is because with so few requests per server there is low contention and Rococo's read-only transactions rarely "abort" (i.e., must continue to another round), while Rococo-SNOW makes the write algorithm more complex. Once contention increases to a moderate level with 30 requests/server, the throughput of Rococo-SNOW matches that of Rococo and then starts to exceed it. In the high-contention workloads that Rococo was designed for, Rococo-SNOW actually has much higher throughput. This is because Rococo-SNOW's read-only transactions always succeed after a single blocking round, while Rococo's read-only transactions often have to retry many times when contention is high.

Figure 7b shows the latency of the read-only stock level transactions, which shows that Rococo-SNOW provides much lower latency than Rococo. Figure 7c shows the "commit" rate of stock level transactions. The low commit rate for these read-only transactions makes them the bottleneck for Rococo under high contention, even though all of Rococo's read-write transactions have a commit rate of 100% (not shown).

7 Related Work

This section reviews existing read-only transactions and discusses other impossibility results.

Existing Read-Only Transactions. Figure 8 categorizes many recent systems with read-only transaction algorithms according to the SNOW properties. Some like Yesquel [1], MySQL Cluster [31], and Spanner [11]'s snapshot reads API are SNOW-optimal and latency-optimal. To be optimal in both the systems must give up one of the power related properties, and all of the three systems give up strict serializability. Spanner's snapshot reads API provides a serializable (but potentially stale) snaphot over the data. Yesquel provides snapshot isolation, which is slightly weaker. MySQL Cluster provides the yet weaker read-committed consistency model.

Many systems are SNOW-optimal but not latency-optimal. These systems made a different design choice and give up a latency related property to be as powerful as possible. One system, Eiger [27], has a bound on the number of rounds needed for read-only transactions. Other non-blocking SNOW-optimal systems have an unbounded number of rounds. These include DrTM [37], RIFL [23], and Sinfonia [2]. The unbounded number of rounds typically comes from algorithms that can abort, e.g., those based on optimistic concurrency control. Rococo-SNOW is a different flavor of SNOW-optimal because it is blocking, as is the algorithm of Spanner-RO [11], which is Spanner's strictly serializable read-only transaction API.

Finally, many systems are neither SNOW-optimal nor latency-optimal. This suggests there is room for improvement in the latency of their read-only transaction algorithms without making a fundamentally different design choice. COPS [26] and Rococo [29] fall into this category, which is the primary reason we developed new algorithms for them. Walter [35], Orbe [14], Chain-Reaction [5], Calvin [36], RAMP [8], Granola [12], TAPIR [38], and Janus [30] also fall in this category and are strong candidates for improvement.

Impossibility Results. Our work is inspired by other impossibility results. The FLP result proves that in a deterministic asynchronous system distributed processes cannot always achieve consensus if even one process can be faulty [16]. FLP is a different type of impossibility result than SNOW because it states a liveness property cannot always be satisfied: it is impossible to guarantee a good thing (consensus) will always eventually happen. In practice, however, consensus despite multiple faulty processes happens regularly. The SNOW theorem, on the other hand, states a safety property cannot always be

System	S	N	O	W
SNOW-optimal and latency-optimal				
Spanner-Snap [11]*	Ser	✓	✓	✓
Yesquel [1]	SI	✓	✓	✓
MySQL Cluster [31]*	RC	✓	✓	✓
SNOW-optimal				
Eiger [27]*	✓	✓	≤ 3	✓
DrTM [37]*	✓	✓	≥ 1	✓
RIFL [23]	✓	✓	≥ 2	✓
Sinfonia [2]	✓	✓	≥ 2	✓
Spanner-RO [11]*	✓	×	✓	✓
Rococo-SNOW*	✓	×	✓	✓
Latency-optimal				
COPS-SNOW*	Causal	✓	✓	×
Neither SNOW-optimal nor latency-optimal				
Janus [30]	✓	×	≤ 2	✓
Calvin [36]	✓	×	2	✓
Rococo [29]*	✓	×	≥ 1	✓
TAPIR [38]*	Ser	×	✓	✓
Granola-Independent [12]*	Ser	✓	≥ 2	✓
Granola-Coordinated [12]*	Ser	✓	≥ 2	✓
Walter [35]	PSI	✓	≤ 2	✓
COPS [26]*	Causal	✓	≤ 2	×
Orbe [14]*	Causal	×	2	×
ChainReaction [5]*	Causal	×	≥ 2	×
RAMP-F [8]*	RA	✓	≤ 2	✓
RAMP-H [8]*	RA	✓	≤ 2	✓
RAMP-S [8]*	RA	✓	2	✓

Figure 8: Categorization of read-only transactions along the SNOW properties. Astericks denote algorithms that are specialized for read-only transactions.

satisfied: it is impossible to guarantee a bad thing (violating strict serializability) will never happen. Any system that can violate a safety property is not safe, and thus cannot be used in practice.

The CAP Theorem proves it is impossible for a distributed data store to always provide consistency (strict serializability) and availability under network partitions [9, 18]. Lipton and Sandberg [25] first discovered and Attiya and Welch [7] later refined a result that shows it is impossible to achieve sequential consistency and low latency in a replicated system. The CAP Theorem and Lipton/Sandberg result are similar to SNOW in that they point to a fundamental design decision for system builders where they must choose some properties at the expense of losing others.

A recent line of work has investigated read-only transaction for transactional memory (TM). Attiya et al. [6] proved that it is impossible to have strictly serializable TM implementations that ensure read-only transactions are *invisible*—i.e., reads do not update memory—and *wait-free*—always terminate regardless of concurrent transactions. Peluso at el. [33] further refined this result with a TM implementation that has wait-free read-only transactions but with a relaxed consistency model. This work explores the possibilities of read-only transactions in a different setting from ours. The concerns of the different setting are different, TM is interested in efficient hardware implementation while we are more interested in a finer granularity of performance properties, e.g., one response.

8 Conclusion

Read-only transactions are a fundamental building block for large-scale applications such as modern Web services. The SNOW Theorem proves that there is a fundamental tradeoff between the power and latency of read-only transactions by showing that it is impossible for an algorithm to provide strict serializability, non-blocking operations, one response per read, and compatibility with write transactions. The resulting notion of SNOW-optimality along with latency-optimality are powerful lenses for examining existing systems and determining if the latency of their read-only transactions can be improved. Using those lenses we designed and implemented COPS-SNOW—a new latency-optimal algorithm—and Rococo-SNOW—a new SNOW-optimal algorithm. Our evaluation demonstrates that both algorithms provide lower latency for read-only transactions.

Acknowledgments

We are grateful to Theano Stavrinos, Minlan Yu, the OSDI program committee, and our shepherd, Dan Ports, for their feedback that improved this work. Our evaluation at scale was made possible by the PRObE testbed, which is supported by NSF awards CNS-1042537 and CNS-1042543. The administrative team for the PRObE testbed went above and beyond in their help to us. This work was supported by NSF grants CNS-1464438, CNS-1514422, and AFOSR grant FA9550-15-1-0302. Part of Christopher Hodsdon's work on this project was done while he was an undergraduate student at Rutgers University-Camden and supported by NSF awards 1218620 and 1433220.

References

[1] AGUILERA, M. K., LENERS, J. B., AND WALFISH, M. Yesquel: scalable SQL storage for Web applications. In *Proc. SOSP* (Oct 2015).

[2] AGUILERA, M. K., MERCHANT, A., SHAH, M., VEITCH, A., AND KARAMANOLIS, C. Sinfonia: A new paradigm forbuilding scalable distributed systems. In *Proc. SOSP* (Oct 2007).

[3] AHAMAD, M., NEIGER, G., KOHLI, P., BURNS, J., AND HUTTO, P. Causal memory: Definitions, implementation, and programming. *Distributed Computing 9*, 1 (1995).

[4] AJOUX, P., BRONSON, N., KUMAR, S., LLOYD, W., AND VEERARAGHAVAN, K. Challenges to adopting stronger consistency at scale. In *Proc. HotOS* (May 2015).

[5] ALMEIDA, S., LEITAO, J., AND RODRIGUES, L. ChainReaction: a causal+ consistent datastore based on chain replication. In *Proc. Eurosys* (Apr 2013).

[6] ATTIYA, H., HILLEL, E., AND MILANI, A. Inherent limitations on disjoint-access parallel implementations of transactional memory. In *Proc. SPAA* (Aug 2009).

[7] ATTIYA, H., AND WELCH, J. L. Sequential consistency versus linearizability. *ACM Trans. Comput. Syst. 12*, 2 (1994).

[8] BAILIS, P., FEKETE, A., GHODSI, A., HELLERSTEIN, J. M., AND STOICA, I. Scalable atomic visibility with RAMP transactions. In *Proc. SIGMOD* (Jun 2014).

[9] BREWER, E. A. Towards robust distributed systems. In *Proc. Principles of Distributed Computing* (Jul 2000).

[10] BRONSON, N., AMSDEN, Z., CABRERA, G., CHAKKA, P., DIMOV, P., DING, H., FERRIS, J., GIARDULLO, A., KULKARNI, S., LI, H., MARCHUKOV, M., PETROV, D., PUZAR, L., SONG, Y. J., AND VENKATARAMANI, V. Tao: Facebook's distributed data store for the social graph. In *Proc. ATC* (Jun 2013).

[11] CORBETT, J. C., DEAN, J., EPSTEIN, M., FIKES, A., FROST, C., ANDSANJAY GHEMAWAT, J. F., GUBAREV, A., HEISER, C., HOCHSCHILD, P., ANDSEBASTIAN KANTHAK, W. H., KOGAN, E., LI, H., LLOYD, A., MELNIK, S., ANDDAVID NAGLE, D. M., QUINLAN, S., RAO, R., ROLIG, L., SAITO, Y., ANDCHRISTOPHER TAYLOR, M. S., WANG, R., AND WOODFORD, D. Spanner: Google's globally-distributed database. In *Proc. OSDI* (Oct 2012).

[12] COWLING, J., AND LISKOV, B. Granola: Low-overhead distributed transaction coordination. In *Proc. ATC* (Jun 2012).

[13] DIXON, P. Shopzilla site redesign: We get what we measure. Velocity Conference Talk, 2009.

[14] DU, J., ELNIKETY, S., ROY, A., AND ZWAENEPOEL, W. Orbe: Scalable causal consistency using dependency matrices and physical clocks. In *Proc. SoCC* (Oct 2013).

[15] DWORK, C., LYNCH, N., AND STOCKMEYER, L. Consensus in the presence of partial synchrony. *Journal of the ACM (JACM) 35*, 2 (1988), 288–323.

[16] FISCHER, M. J., LYNCH, N. A., AND PATERSON, M. S. Impossibility of distributed consensus with one faulty process. In *Proc. Principles of Database Systems* (Mar 1983).

[17] GIBSON, G., GRIDER, G., JACOBSON, A., AND LLOYD, W. Probe: A thousand-node experimental cluster for computer systems research. *USENIX ;login:* (June 2013).

[18] GILBERT, S., AND LYNCH, N. Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services. In *ACM SIGACT News* (Jun 2002).

[19] HERLIHY, M. P., AND WING, J. M. Linearizability: A correctness condition for concurrent objects. *ACM Transactions on Program-ming Languages and Systems 12*, 3 (1990), 463–492.

[20] KALLMAN, R., KIMURA, H., NATKINS, J., PAVLO, A., RASIN, A., ZDONIK, S., JONES, E. P., MADDEN, S., STONEBRAKER, M., ZHANG, Y., ET AL. H-store: a high-performance, distributed main memory transaction processing system. *Proceedings of the VLDB Endowment 1*, 2 (2008), 1496–1499.

[21] LAMPORT, L. Time, clocks, and the ordering of events in a distributed system. *Commun. ACM 21*, 7 (1978).

[22] LAMPORT, L. How to make a multiprocessor computer that correctly executes multiprocess programs. *IEEE Trans. Comput.* (1979).

[23] LEE, C., PARK, S. J., KEJRIWAL, A., MATSUSHITAY, S., AND OUSTERHOUT, J. Implementing linearizability at large scale and low latency. In *Proc. SOSP* (Oct 2015).

[24] LINDEN, G. Make data useful. Stanford CS345 Talk, 2006.

[25] LIPTON, R. J., AND SANDBERG, J. S. PRAM: A scalable shared memory. Tech. Rep. TR-180-88, Princeton Univ., Dept. Comp. Sci., 1988.

[26] LLOYD, W., FREEDMAN, M. J., KAMINSKY, M., AND ANDERSEN, D. G. Don't settle for eventual: Scalable causal consistency for wide-area storage with COPS. In *Proc. SOSP* (Oct 2011).

[27] LLOYD, W., FREEDMAN, M. J., KAMINSKY, M., AND ANDERSEN, D. G. Stronger semantics for low-latency geo-replicated storage. In *Proc. NSDI* (Apr 2013).

[28] LU, H., VEERARAGHAVAN, K., AJOUX, P., HUNT, J., SONG, Y. J., TOBAGUS, W., KUMAR, S., AND LLOYD, W. Existential consistency: Measuring and understanding consistency at facebook. In *Proc. SOSP* (Oct 2015).

[29] MU, S., CUI, Y., ZHANG, Y., LLOYD, W., AND LI, J. Extracting more concurrency from distributed transactions. In *Proc. OSDI* (Oct 2014).

[30] MU, S., NELSON, L., LLOYD, W., AND LI, J. Consolidating concurrency control and consensus for commits under conflicts. In *Proc. OSDI* (Nov 2016).

[31] MYSQL. MySQL :: MySQL Cluster CGE. `https://www.mysql.com/products/cluster/`, 2016.

[32] PAPADIMITRIOU, C. H. The serializability of concurrent database updates. *Journal of the ACM 26*, 4 (1979).

[33] PELUSO, S., PALMIERI, R., ROMANO, P., RAVINDRAN, B., AND QUAGLIA, F. Disjoint-access parallelism: Impossibility, possibility, and cost of transactional memory implementations. In *Proc. PODC* (Jul 2015).

[34] SCHURMAN, E., AND BRUTLAG, J. The user and business impact of server delays, additional bytes, and HTTP chunking in web search. Velocity Conference Talk, 2009.

[35] SOVRAN, Y., POWER, R., AGUILERA, M. K., AND LI, J. Transactional storage for geo-replicated systems. In *Proc. SOSP* (Oct 2011).

[36] Thomson, A., Diamond, T., Weng, S.-C., Ren, K., Shao, P., and Abadi, D. J. Calvin: Fast distributed transactions for partitioned database systems. In *Proc. SIGMOD* (May 2012).

[37] Wei, X., Shi, J., Chen, Y., Chen, R., and Chen, H. Fast in-memory transaction processing using RDMA and HTM. In *Proc. SOSP* (Oct 2015).

[38] Zhang, I., Sharma, N. K., Szekeres, A., Krishnamurthy, A., and Ports, D. R. K. Building consistent transactionswith inconsistent replication. In *Proc. SOSP* (Oct 2015).

Correlated Crash Vulnerabilities

Ramnatthan Alagappan, Aishwarya Ganesan, Yuvraj Patel,
Thanumalayan Sankaranarayana Pillai, Andrea C. Arpaci-Dusseau, Remzi H. Arpaci-Dusseau
University of Wisconsin – Madison

Abstract

Modern distributed storage systems employ complex protocols to update replicated data. In this paper, we study whether such update protocols work correctly in the presence of *correlated crashes*. We find that the correctness of such protocols hinges on how local file-system state is updated by each replica in the system. We build PACE, a framework that systematically generates and explores persistent states that can occur in a distributed execution. PACE uses a set of generic rules to effectively prune the state space, reducing checking time from days to hours in some cases. We apply PACE to eight widely used distributed storage systems to find *correlated crash vulnerabilities*, i.e., problems in the update protocol that lead to user-level guarantee violations. PACE finds a total of 26 vulnerabilities across eight systems, many of which lead to severe consequences such as data loss, corrupted data, or unavailable clusters.

1 Introduction

Modern distributed storage systems are central to large scale internet services [19,22,28,66]. Important services such as photo stores [74,77,79], e-commerce [61], video stores [27], text messaging [82], social networking [93], and source repositories [78] are built on top of modern distributed storage systems. By providing replication, fault tolerance, high availability, and reliability, distributed storage systems ease the development of complex software services [17,31,59,75].

Reliability of user data is one of the most important tenets of any storage system [3,41,49,69]. Distributed storage systems improve reliability by replicating data over a collection of servers [7,13,63,80,84,89,91].

To safely replicate and persist application data, modern storage systems implement complex data update protocols. For example, ZooKeeper [5] implements an atomic broadcast protocol and several systems including LogCabin [57] and etcd [23] implement the Raft consensus protocol to ensure agreement on application data between replicas. Although the base protocols (such as atomic broadcast [12], Raft [68], or Paxos [51]) are provably correct, implementing such a protocol without bugs is still demanding [16, 36, 42, 44, 94], especially when machines can crash at any instant [53].

Many distributed storage systems can recover from single node or partial cluster failures. In this study, we consider a more insidious crash scenario in which all replicas of a particular data shard crash at the same time and recover at a later point. We refer to such crash scenarios as *correlated failures*. Correlated failures are common and several instances of such failures have been reported in the recent past [11,20,21,25,26,35,43,49,65, 92]; these failures occur due to root causes such as data-center-wide power outages [38], operator errors [97], planned machine reboots, or kernel crashes [35].

When nodes recover from a correlated failure, the common expectation is that the data stored by the distributed system would be recoverable. Unfortunately, local file systems (which store the underlying data and metadata of many distributed storage systems) complicate this situation. Recent research has shown that file systems vary widely with respect to how individual operations are persisted to the storage medium [70]. For example, testing has revealed that in ext4, f2fs [8], and u2fs [58], one cannot expect the following guarantee: a file always contains a prefix of the data appended to it (i.e., no unexpected data or garbage can be found in the appended portion) after recovering from a crash. The same test also shows that this property *may* be held by btrfs and xfs. Since most practical distributed systems run atop local file systems [47,62,81], it is important for them to be aware of such behaviors. These file-system nuances can result in unanticipated persistent states in one or more nodes when a distributed storage system recovers from a correlated crash.

Recent studies [14, 70] have demonstrated that these widely varying file-system behaviors increase the complexity of building a crash-consistent update protocol, even for single machine applications such as SQLite. We refer to this form of crash consistency as *single-machine (application-level) crash consistency*.

Distributed storage systems have to deal with the complexity of building a crash-consistent storage update protocol in addition to correctly implementing a distributed agreement and recovery protocol. We refer to this form of crash consistency in a distributed setting as *correlated crash consistency*. Although the challenges of building a crash-consistent distributed update protocol have the same flavor as building a crash-consistent single-machine protocol, correlated crash consistency is a fundamentally different problem for three reasons.

First, distributed systems can fail in more ways than a single machine system. Since a distributed system constitutes many components, a group of components may fail together at the same or different points in the protocol. Second, unique opportunities and problems exist in distributed crash recovery; after a failure, it is possible for one node in an inconsistent state to repair its state by contacting other nodes or to incorrectly propagate the corruption to other nodes. In contrast, single-machine applications rarely have external help. Third, crash recovery in a distributed setting has many more possible paths than single-machine crash recovery as distributed recovery depends on states of many nodes in the system.

We say a distributed system has a *correlated crash vulnerability* if a correlated crash during the execution of the system's update protocol (and subsequent recovery) exposes a user-level guarantee violation. In this paper, we examine whether distributed storage systems are vulnerable to correlated crashes. To do this, we introduce PACE, a novel framework that systematically explores correlated crash states that occur in a distributed execution.

PACE considers consistent cuts in the distributed execution and generates persistent states corresponding to those cuts. PACE models local file systems at individual replicas using an *abstract persistence model* (APM) [70] which captures the subtle crash behaviors of a particular file system. PACE uses *protocol-specific knowledge* to reduce the exploration state space by systematically choosing a subset of nodes to introduce file-system nuances modeled by the APM. In the worst case, if no attributes of a distributed protocol are known, PACE can operate in a slower brute-force mode to still find vulnerabilities.

We applied PACE to eight widely used distributed storage systems spanning important domains including database caches (Redis [76]), configuration stores (ZooKeeper [5], LogCabin [57], etcd [23]), real-time databases (RethinkDB [83]), document stores (MongoDB [60]), key-value stores (iNexus [46]), and distributed message queues (Kafka [6]).

We find that many of these systems are vulnerable to correlated crashes. Modern distributed storage systems expect certain guarantees from file systems such as ordered directory operations and atomic appends for their local update protocols to work correctly. We also find that in many cases global recovery protocols do not use intact replicas to fix the problematic nodes. PACE found a total of 26 vulnerabilities that have severe consequences such as data loss, silent corruption, and unavailability. We also find that many vulnerabilities can be exposed on commonly used file systems such as ext3, ext4, and btrfs. We reported 18 of the discovered vulnerabilities to application developers. Twelve of them have been already fixed or acknowledged by developers. While some vulnerabilities can be fixed by straightforward code changes, some are fundamentally hard to fix.

Our study also demonstrates that PACE is *general*: it can be applied to any distributed system; PACE is *systematic*: it explores different systems using general rules that we develop; PACE is *effective*: it found 26 *unique* vulnerabilities across eight widely used distributed systems. PACE's source code and details of the discovered vulnerabilities are publicly available [2].

The rest of the paper is organized as follows. We first describe correlated crash consistency in detail (§2). Next, we explain how PACE works and how it uses protocol-awareness to systematically reduce exploration state space (§3). Then, we present our study of correlated crash vulnerabilities in distributed storage systems (§4). Finally, we discuss related work (§5) and conclude (§6).

2 Correlated Crash Consistency

Building a crash-consistent distributed update protocol is complex for two reasons: machines can crash at any time in a correlated fashion and updates to the local file system have to be performed carefully to recover from such crashes. Given this complexity, we answer the following question in this paper: *Do modern distributed storage systems implement update and recovery protocols that function correctly when nodes crash and recover in a correlated fashion, or do they have vulnerabilities*? To answer this question, we first describe the failure model that we consider and build arguments for why the considered failure model is important. Next, we explain the system states we explore to find if a distributed update protocol has correlated crash vulnerabilities.

2.1 Failure Model

Components in a distributed system can fail in various ways [39]. Most practical systems do not deal with Byzantine failures [52] where individual components may give conflicting information to different parts of the system. However, they handle fail-recover failures [39] where components can crash at any point in time and recover at any later point in time, after the cause of the failure has been repaired. When a node crashes and recovers, all its in-memory state is lost; the node is left only with its persistent state. Our study considers only such fail-recover failures.

A common expectation is that practical distributed systems would not violate user-level guarantees in the face of such fail-recover failures. However, nuances in local file system can cause unanticipated persistent states to arise after a crash and a subsequent reboot, complicating proper recovery. We explain such nuances in local file-system behaviors later (§2.2.2). Notice that this is how one individual node crashes and recovers; other nodes may continue to function and make progress.

Our failure model concentrates on a more devastating scenario where *all* or a *group* of nodes crash and recover together. We refer to this type of failure as a correlated crash. Specifically, we focus on two kinds of correlated crash scenarios: first, data-center-wide power outages where *all* machines in the cluster crash and recover together; second, correlated failures where only a *group of machines* containing all replicas for a shard of data crash and recover together and other machines (that are not part of the shard) in the cluster do not react to the failure.

Large-scale correlated failures such as cluster-wide power outages are common and occur in the real world [11,20,21,25,26,35,43,49,65,92]. For example, a recent study from Google [35] showed that node failures in Google data centers are often correlated and the causes of node failures fall into three categories: application restarts, planned machine reboots, and unplanned machine reboots. From data over a period of three months, the study showed that as many as 15 unplanned reboots and 30 planned reboots per 1000 machines can happen in a single day. Also, a *failure burst* in one instance of a distributed file system [37] can take down as many as 50 machines at almost the same time. This kind of failure typically can be seen during a power outage in a data center. Rolling kernel upgrades also cause failure bursts that can take down around 20 machines within a short window of time.

Although the system cannot progress when all replicas crash, the common expectation is that the data stored by the storage system will be recoverable after the replicas come alive (for example, after power has been restored).

Our failure model is not intended to reason about scenarios where only a subset of replicas of a particular data shard crash and recover by themselves. Also, the vulnerabilities we find with our correlated failure model do not apply to a geo-replicated setting; in such a setting, conscious decisions place replicas such that one power failure cannot affect all replicas at the same time. While correlated failures are less problematic in such settings, the storage systems we examine in this study are heavily tested and the common expectation is that these systems should be reliable irrespective of how they are deployed

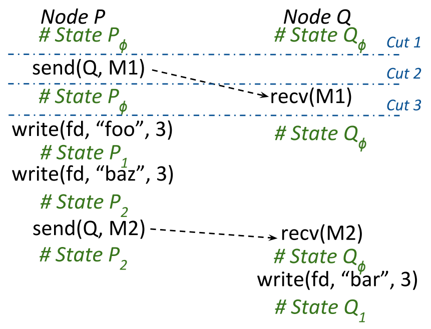


Figure 1: **A simple distributed protocol.** *The figure shows a simple distributed protocol. Annotations show the persistent state after performing each operation. Dash dot lines show different cuts.*

and the probability of failures. Further, many deployments are not geo-replicated and thus may expect strong guarantees even in the presence of correlated crashes. Overall, crash-correctness should be deeply ingrained in these systems regardless of deployment decisions.

2.2 Distributed Crash States

Now we explain the global system states that result due to correlated crashes. As we explained, after a crash and subsequent reboot, a node is left only with its persistent data. The focus of our study is in checking only the resulting *persistent states* when failures happen. The global states that we capture are similar to distributed snapshots [18] described by Chandy and Lamport. The main difference between a generic distributed snapshot and a global persistent state is that the latter consists only of the on-disk state and not the in-memory state of the machines. Moreover, since network channels do not affect persistent on-disk state, our global persistent states do not keep track of them.

To understand the persistent states that we capture, consider a cluster of three machines named A, B, and C. Assume that the initial persistent states of these machines are A_ϕ, B_ϕ, and C_ϕ, respectively. Assume that a workload W run on this cluster transitions the persistent states to A_f, B_f, and C_f, respectively. For instance, W could be a simple workload that inserts a new key-value pair into a replicated key-value store running on A, B, and C. Notice that the persistent state of all nodes goes through a transition before arriving at the final states A_f, B_f, and C_f. A correlated crash may happen at any time while W runs, and after a reboot, the persistent state of a node X may be any intermediate state between X_ϕ and X_f where X can be A or B or C. For simplicity, we refer to this collection of persistent states across all nodes as global persistent state or simply global state. If a particular global state G can occur in an execution, we call G a *reachable* global state.

2.2.1 Reachable Global States

The reachability of a global state depends on two factors: the order in which messages are exchanged between nodes and the local file systems of the nodes. To illustrate the first factor, consider a distributed protocol shown in Figure 1. In this protocol, node P starts by sending message `M1`, then writes `foo` and `baz` to a file, and then sends another message `M2` to node Q. Node Q receives `M1` and `M2` and then writes `bar` to a file. For now, assume that the toy application is single threaded and all events happen one after the other. Also assume that the file system at P and Q is synchronous (i.e., operations are persisted *in order* to the disk). We will soon remove the second assumption and subsequently the first (§3.2).

Assume that the initial persistent state of P was P_ϕ and Q was Q_ϕ. After performing the first and second write, P moves to P_1 and P_2, respectively. Similarly, Q moves to Q_1 after performing the write. Notice that $<P_\phi, Q_\phi>$ is a reachable global persistent state as P could have crashed before writing to the file and Q could have crashed before or after receiving the first message. Similarly, $<P_2, Q_1>$ and $<P_2, Q_\phi>$ are globally reachable persistent states.

In contrast, $<P_\phi, Q_1>$ and $<P_1, Q_1>$ are unreachable persistent states as it is not possible for Q to have written the file without P sending the message to it. Intuitively, global states that are not reachable in an execution are logically equivalent to inconsistent cuts in a distributed system [10]. For example, $<P_\phi, Q_1>$ and $<P_1, Q_1>$ are inconsistent cuts since the `recv` of `M2` is included in the cut but the corresponding `send` is excluded from the cut. Also, network operations such as `send` and `recv` do not affect the persistent state. For example, the three different cuts shown in Figure 1 map onto the same persistent state $<P_\phi, Q_\phi>$.

Next, we consider the fact that the local file systems at P and Q also influence the global states. Assume that the application is still single threaded but writes issued by an application can be buffered in memory as with modern file systems. Depending on which exact file system and mount options are in use, modern file systems may reorder some (or many) updates [9, 70, 73]. With this asynchrony and reordering introduced by the file system, it is possible for the second write `baz` to reach the disk before the first write `foo`. Also, it is possible for P to crash after `baz` is persisted and the message is sent to Q, but before `foo` reaches the disk. In such a state of P, it is possible for Q to have either reached its final state Q_1 or crash before persisting `bar` and so remain in Q_ϕ. All these states are globally reachable.

2.2.2 File-system Behavior

The reordering of writes by the file system is well understood by experienced developers. To avoid such reordering, developers force writes to disk by carefully issuing `fsync` on a file as part of the update protocol. Although some common behaviors such as reordering of writes are well understood, there are subtle behaviors that application developers find hard to reason about. For example, the following subtle behavior is not well documented: if a crash happens when appending a single block of data to a file in ext4 writeback mode, the file may contain garbage on reboot. These behaviors are neither bugs nor intended features, but rather implications of unrelated performance improvements. To worsen the problem, these subtle behaviors vary across file systems.

Recent research [4,14,70–72] classifies file-system behaviors into two classes of properties: atomicity and ordering. The atomicity class of properties say whether a particular file system must persist a particular operation in an atomic fashion in the presence of crashes. For instance, must ext2 perform a `rename` in an atomic way or can it leave the system in any intermediate state? The ordering class of properties say whether a particular file system must persist an operation A before another operation B. For instance, must ext4 (in its default mode) order a `link` and a `write` operation? While ext4 orders directory operations and file write operations, the same does not hold true with btrfs which can reorder directory operations and write operations.

Given these variations across file systems and sometimes even across different configurations of the same file system, it is onerous to implement a crash-consistent protocol that works correctly on all file systems. Recent research has discovered that single-machine applications have many vulnerabilities in their update protocols which can cause them to corrupt or lose user data [4, 70, 98].

Distributed storage systems also face the same challenge as each replica uses its local file system to store user data and untimely crashes may leave the application in an inconsistent state. However, distributed systems have more opportunities for recovery as redundant copies of data exist on other nodes.

3 Protocol-Aware Crash Explorer

To examine if distributed storage systems violate user-level guarantees in correlated crash scenarios, we build a generic correlated crash exploration framework, PACE, which systematically generates persistent states that can occur in a distributed execution in the presence of correlated crashes. We note here that PACE is not intended to catch bugs in distributed consensus protocols. Specifically, it does not exercise reordering of network messages to explore corner cases in consensus protocols; as explained later (§5), distributed model checkers attack this problem. PACE's intention is to examine the interaction of global crash recovery protocols and the nuances in local storage protocols (introduced by each replica's local file system), in the presence of correlated crashes.

Some vulnerabilities that we discover are exposed only if a particular file-system operation is reordered on all replicas while some vulnerabilities are exposed even when the reordering happens on a single replica. Using observations from how vulnerabilities are exposed and a little knowledge about the distributed protocol, we make our exploration *protocol-aware*. Using this awareness, PACE can prune the search space while finding as many vulnerabilities as a brute-force search. To explain how protocol-aware exploration works, we first describe the design of our crash exploration framework.

3.1 Design and Implementation Overview

PACE is easy to use and can be readily applied to any distributed storage system. PACE needs a workload script and a checker script as inputs. For many modern distributed systems, a group of processes listening on different ports can act as a cluster of machines. For systems that do not allow this convenience, we use a group of Docker [30] containers on the same machine to serve as the cluster. In either case, PACE can test the entire system on a single machine. PACE is implemented in around 5500 lines of code in Python.

To begin, PACE starts the cluster with system call tracing, runs the workload, and then stops the cluster after the workload is completed. PACE parses the traces obtained and identifies cross node dependencies such as a `send` on one node and the corresponding `recv` on some other node. After the traces are parsed and cross node dependencies established, PACE replays the trace to generate different persistent crash states that can occur in the traced execution. A system-specific checker script is run on top of each crash state; the checker script asserts whether user-level guarantees (e.g., committed data should not be corrupted or lost) hold. Any violations in such assertions are reported as vulnerabilities. We next discuss what correlated crash states can occur in a distributed execution and how we generate them.

3.2 Crash States

We use a running example of a ZooKeeper cluster executing an update workload for further discussion. PACE produces a diagrammatic representation of the update protocol as shown in Figure 2.

First, the client contacts the leader in the ZooKeeper cluster. The leader receives the request and orchestrates the atomic broadcast protocol among its followers as shown by `send` and `recv` operations and careful updates to the file system (`write` and `fdatasync` on a log file that holds user data). Finally, after ensuring that the updated data is carefully replicated and persisted, the client is acknowledged. At this point, it is guaranteed that the data will be consistent and durable.

Note that each node runs multiple threads and the figure shows the observed order of events when the traces were collected. If arbitrary delays were introduced, the order may or may not change, but this observed order is one schedule among all such possible schedules.

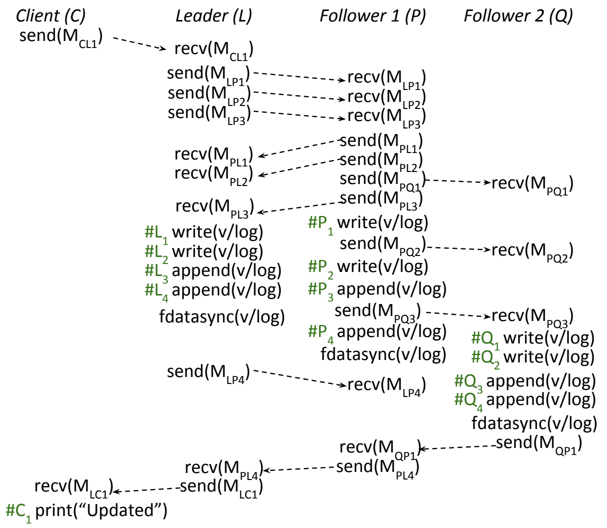


Figure 2: **ZooKeeper protocol for an update workload.** *The figure shows the sequence of steps when the client interacts with the ZooKeeper cluster. The workload updates a value. The client prints to stdout once the update request is acknowledged.*

We reiterate here that PACE captures crash states that occur due to a correlated failure where all replicas fail together. PACE is not intended to reason about partial crashes where only a subset of replicas crash.

3.2.1 Globally Reachable Prefixes

Assume that all nodes shown in Figure 2 start with persistent state X_ϕ where X is the node identifier with L for leader, C for client, and so forth. M_{XYi} is the i^{th} message sent by X to Y. All operations that affect persistent state are annotated with the persistent state to which the node transitions by performing that operation. For example, the leader transitions to state L_1 after the first `write` to a file. The total set of global persistent states is the cross product of all local persistent states. Precisely, the total set is the cross product of the sets $\{C_\phi, C_1\}$, $\{L_\phi, L_1, L_2, L_3, L_4\}$, $\{P_\phi, P_1, P_2, P_3, P_4\}$ and $\{Q_\phi, Q_1, Q_2, Q_3, Q_4\}$. However, some of the global states in this resultant set cannot occur in the distributed execution. For example, $<C_\phi, L_2, P_2, Q_1>$ is an inconsistent cut and cannot occur as a global state since it is not possible for Q to receive M_{PQ3} before P reaches P_3 and then sends M_{PQ3}.

We refer to a global state that is reachable in this trace as a globally reachable persistent prefix or simply *globally reachable prefix*. We call this a prefix as it is a prefix of the file-system operations within each node.

Previous work [70] has developed tools to uncover single-machine crash vulnerabilities. Such tools trace only file-system related system calls and do not trace network operations. Hence, they cannot capture dependencies across different nodes in a distributed system. Such tools cannot be directly applied to distributed systems; if applied, they may generate states that may not actually occur in a distributed execution and thus can report spurious vulnerabilities. On the other hand, PACE captures all cross node dependencies and so generates only states that can occur in a distributed execution.

3.2.2 File-system Persistence Models

Generating globally reachable prefixes does not require any knowledge about how a particular file system persists operations. As we discussed, file systems exhibit important behaviors with respect to how operations are persisted. We borrow the idea of *abstract persistence model (APM)* from our previous work [70] to model the file system used by each node.

An APM specifies all constraints on the atomicity and ordering of file-system operations for a given file system, thus defining which crash states are possible. For example, in an APM that specifies the ext2 file system, appends to a file can be reordered and the `rename` operation can be split into smaller operations such as deleting the source directory entry and creating the target directory entry. In contrast, in the ext3 (data-journaling) APM, appends to a file cannot be reordered and the `rename` operation cannot be split into smaller operations. An APM for a new file system can be easily derived using the *block order breaker* (BOB) tool [70].

PACE considers all consistent cuts in the execution to find globally reachable prefixes. On each such globally reachable prefix, PACE applies the APM (that specifies what file-system specific crash states are possible) to produce more states. The default APM used by PACE has few restrictions on the possible crash states. Intuitively, our default APM models a file system that provides the least guarantees when crashes occur but is still `POSIX` compliant. For simplicity, we refer to file-system related system calls issued by the application as *logical operations* and the smaller operations into which each logical operation is broken down as *micro operations*. We now describe our default APM.

Atomicity of operations. Applications may require a single logical operation such as *append* or *overwrite* to be atomically persisted for correctness. In the default APM used by PACE, all logical operations are broken into the following micro operations: *write_block*, *change_size*, *create_dir_entry*, and *delete_dir_entry*. For example, a logical truncate of a file will be broken into *change_size* followed by *write_block(random)* followed by *write_block(zeroes)*. Similarly, a rename will be broken into *delete_dir_entry(dest)* + *truncate if last link* followed by *create_dir_entry(dest)* followed by *delete_dir_entry(src)*. Overwrites, truncates, and appends are split into micro operations aligned at the block boundary or simply into three micro operations. PACE can generate crash states corresponding to different intermediate states of the logical operation.

Ordering between operations. Applications may require that a logical operation A be persisted before another logical operation B for correctness. To reorder operations, PACE considers each pair of operations (A, B) and applies all operations from the beginning of the trace until B except for A. This reordering produces a state corresponding to the situation where the node crashes after all operations up to B have been persisted but A is still not persisted. The ordering constraint for our default APM is as follows: all operations followed by an *fsync* on a file or directory F are ordered after the operations on F that precede the *fsync*.

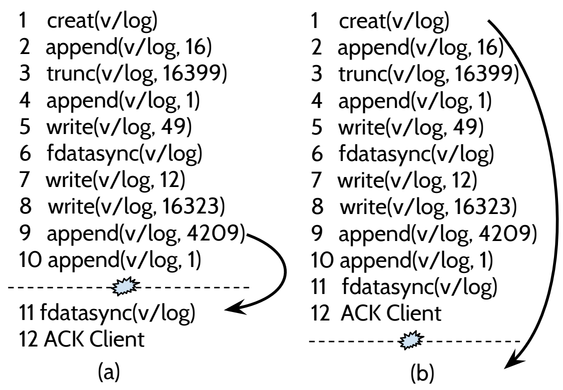
We now describe how applying an APM produces more states on a *single* machine. Consider the ZooKeeper protocol in which $<C_\phi, L_1, P_2, Q_\phi>$ is a globally reachable prefix. P has moved to P_2 by applying two `write` operations starting from its initial state P_ϕ. On applying the default APM onto the above prefix, PACE recognizes that on node P it is possible for the second write to reach the disk before the first one (by considering different ordering between two operations). Hence, it can *reorder* the first write after the second write on P. This resultant state is different from the prefix. In this resultant state, after recovery, P will see a file-system state where the second write to the log is persisted but effects of the first write are missing. If there were an `fsync` or `fdatasync` after the first write, then the default APM cannot and will not reorder the two write operations. This reordering is within a *single* node; similar reorderings can be exercised on all nodes.

Depending on the APM specification, logical operations can be partially persisted or reordered or both at each node in the system. Intuitively, applying an APM on a global prefix *relaxes* its constraints. This relaxation allows the APM to partially persist logical operations (atomicity) or reorder logical operations with one another (ordering). We refer to the relaxations allowed by an APM as *APM-allowed relaxations* or simply *APM relaxations*. For simplicity, we refer to this process of relaxing the constraints (by reordering and partially persisting operations) as applying that particular relaxation.

PACE can be configured with any APM. We find the most vulnerabilities with our default and ext2 APMs. We also report the vulnerabilities when PACE is configured with APMs of other commonly used file systems.

3.3 Protocol-Aware Exploration

While applying relaxations on a single node results in many persistent states for that node, PACE needs to consider applying different relaxations across every combination of nodes to find vulnerabilities. As a consequence, there are several choices for how PACE can apply relaxations. Consider a five node cluster and assume that n relaxations are possible in one node. Then, as-



```
1  creat(v/log)
2  append(v/log, 16)
3  trunc(v/log, 16399)
4  append(v/log, 1)
5  write(v/log, 49)
6  fdatasync(v/log)
7  write(v/log, 12)
8  write(v/log, 16323)
9  append(v/log, 4209)
10 append(v/log, 1)
-----------------------
11 fdatasync(v/log)
12 ACK Client
(a)
```

```
1  creat(v/log)
2  append(v/log, 16)
3  trunc(v/log, 16399)
4  append(v/log, 1)
5  write(v/log, 49)
6  fdatasync(v/log)
7  write(v/log, 12)
8  write(v/log, 16323)
9  append(v/log, 4209)
10 append(v/log, 1)
11 fdatasync(v/log)
12 ACK Client
-----------------------
(b)
```

Figure 3: **Local file-system update protocol on a single ZooKeeper node.** *The figure shows the sequence of file-system operations on a single ZooKeeper node. Operations 1 through 6 happen on node initialization and operations 7 through 12 when the client starts interacting. Several operations that happen on initialization are not shown for clarity. (a) and (b) show two different crash scenarios.*

suming there are no cross node dependencies, there are $\binom{5}{1} * n + \binom{5}{2} * n^2 + \binom{5}{3} * n^3 + \binom{5}{4} * n^4 + \binom{5}{5} * n^5$ ways of combining the relaxations across nodes. Even for a moderate n such as 20, there are close to 4 million states. A brute-force approach would explore all such states. We now explain how PACE prunes this space by using knowledge about the distributed protocols (such as agreement and leader election) employed by a system.

3.3.1 Replicated State Machine Approaches

We use the same ZooKeeper traces shown in Figure 2 for this discussion. For simplicity, we assume that there are odd number of nodes in the system.

ZooKeeper implements an atomic broadcast protocol which is required to run a replicated state machine (RSM) [45, 68, 90]. There are various paradigms to implement an RSM some of which include Paxos [51], Raft [68], and atomic broadcast [24]. Google's Chubby [15] implements a Paxos-like algorithm and LogCabin [57] implements Raft. An RSM system as a whole should continue to make progress as long as a *majority* of the nodes are operational and can communicate with each other and the clients [68].

Figure 3(a) shows the file-system operations on a single ZooKeeper node; network operations are not shown for clarity. The tenth operation appends one byte to the log to denote the commit of a transaction after which the file is forced to disk by the `fdatasync` call. It is possible for the tenth operation to reach the disk *before* the ninth operation and a crash can happen at this exact point before the `fdatasync` call. After this crash and subsequent restart, ZooKeeper would fail to start as it detects a checksum mismatch for the data written, and the node becomes unusable. The same reordering can happen on all nodes, rendering the entire cluster unusable.

In the simple case where this reordering happens on only one node, even though that single node would fail to start, the other two nodes still constitute a *majority* and so can elect a leader and make progress. PACE uses this knowledge about the protocol to eliminate testing cases where a reordering happens on only one node. Also, it is unnecessary to apply the relaxation on all three nodes as the cluster can become unavailable even when the relaxation is applied on just a majority (any two) of the nodes.

As another example, consider the same protocol but with a different crash that happens after the client is acknowledged, as shown in Figure 3(b). Once acknowledged, ZooKeeper guarantees that the data is replicated and persisted to disk on a majority of nodes. The directory entry for the log file has to be persisted explicitly by performing an `fsync` on the parent directory [1, 70] to ensure that the log file is present on disk even after a crash. However, ZooKeeper does not `fsync` the parent directory and so it is possible for the log file to go missing after a crash. On a single node, if the log file is lost, it does not lead to user-visible global data loss as the majority still has the log file. Similar to the unavailability case, a global data loss can happen if the same reordering happens on a majority of nodes even if the data exists on one other node where this reordering did not happen.

Thus, we observe that in any RSM system, it is required that a particular APM relaxation is applied on at least a majority of nodes for a vulnerability to be exposed globally. Also, it is unnecessary to apply an APM relaxation on all possible majority choices; for example, in a system with five nodes, applying a relaxation on three, four, or five nodes (all of which represent a majority) will expose the same vulnerability. This knowledge is not system-specific, but rather protocol-specific.

System-independent. LogCabin is a system similar to ZooKeeper that provides a configuration store on top of the consensus module but uses the Raft protocol to implement an RSM. When applying a particular APM relaxation, LogCabin can lose data. For this data loss vulnerability to be exposed, the relaxation has to be applied on at least a majority of the nodes. This observation is not specific to a particular system; rather, it holds true across ZooKeeper and LogCabin because both systems are RSM protocol implementations.

Using our observation, we derive the following rule that helps PACE eliminate a range of states: *For any RSM system with N replicas, check only states that would result when a particular APM relaxation is applied on an exact majority (where exactly $\lceil N/2 \rceil$ servers are chosen from N) of the nodes*. Note that there are $\binom{N}{\lceil N/2 \rceil}$ ways of choosing the exact majority.

We note that the pruning rule does not guarantee finding all vulnerabilities. It works because it makes an important assumption: the base consensus protocol is implemented correctly. PACE is not intended to catch bugs in consensus protocol implementations.

We now make a further observation about RSM protocols that can further reduce the state space. Consider the data loss vulnerability shown in Figure 3(b). Surprisingly, sometimes a global data loss may *not* be exposed even when the reordering happens on a majority. To see why, consider that the current leader (L) and the first follower (P) lose the log file as the *creat* operation is not persisted before the crash. In this case, the majority has lost the file. On recovery, the possibility of global data loss depends on *who is elected as the leader the next time*. Specifically, the data will be lost, if either L or P is elected as the new leader. On the other hand, if the second follower Q is elected as the leader, then the data will not be lost. In effect, the data will be lost if a node that lost its local data becomes the leader the next time, irrespective of the presence of the same data on other nodes.

In Raft, on detecting an inconsistency, the followers are forced to duplicate the leader's log (i.e., the log entries flow only outward from the leader) [68]. This enforcement is required to satisfy safety properties of Raft. While ZooKeeper's atomic broadcast (ZAB) does not explicitly specify if the log entries only flow outward from the leader, our experiments show that this is the case. Previous work also supports our observation [68].

This brings a question that counters our observation: *Why not apply the relaxation on any one node and make it the leader during recovery*? Consider the reordering shown in Figure 3(b). If this reordering happens on one node, that node will lose the log; it is *not* possible for this node to be elected the leader as other nodes would notice that this node has missing log entries and not vote for it. If this node is not elected the leader, then local data loss would not result in global data loss.

In contrast, if the log is lost on two nodes, the two nodes still constitute a majority and so one of them can become the leader and therefore override the data on the third node causing a global data loss. However, it is possible for the third node Q, where the data was not lost, to become the leader and so hide the global data loss.

Given this information, we observe that it is required only to check states that result from applying a particular APM relaxation on *any one* exact majority of the nodes. In a cluster of five nodes, there are $\binom{5}{3} = 10$ ways of choosing an exact majority and it is enough to check any one combination from the ten. To effectively test if a global vulnerability can be exposed, we strive to enforce the following: *when the cluster recovers from a crashed state, if possible, the leader should be elected from the set of nodes where the APM relaxation was applied.* Sometimes the system may constrain us from enforcing this; however, if possible, we enforce it automatically to drive the system into vulnerable situations.

From the two observations, we arrive at two simple, system-independent, and protocol-aware exploration rules employed by PACE to prune the state space and effectively search for undesired behaviors:

- **R1**: *For any RSM system with N servers where followers duplicate leader's log, generate states that would result if a particular APM relaxation is applied on any exact majority of the servers.*
- **R2**: *For all states generated using R1, if possible, enforce that the leader is elected from exact majority in which the APM relaxation was applied.*

Since we did not see popular practical systems that use RSM approaches where log entries can flow in both directions like in Viewstamped replication [55, 67] or where there can be multiple proposers at the same time like in Paxos, we have not listed the rules for them. We leave this generalization as an avenue for future work.

3.3.2 Other Replication Schemes

PACE also handles replicated systems that do not use RSM approaches: Redis, Kafka, and MongoDB. Applications belonging to this category do not strictly require a majority for electing a leader and committing transactions. For example, in Redis' default configuration, the master is fixed and cannot be automatically re-elected by a majority of slaves if the master fails. Moreover, it is possible for the master to make progress without the slaves. Similarly, Kafka maintains a metadata structure called the *in-sync replicas* and any node in this set can become the leader without consent from the majority.

Systems belonging to this category typically force slaves to sync data from the master. Hence, any problem in the master can easily propagate to the slaves. This hints that applying APM relaxations on the master is necessary. Next, since our workloads ensure that the data is synchronously replicated to all nodes, it is unacceptable to read stale data from the slaves once an acknowledgment is received. This hints that applying APM relaxations on any slave and subsequent reads from the slave can expose the stale data problem. Since systems of this type can make progress even if one node is up, we need to apply APM relaxations on all the nodes to expose cluster unavailability vulnerabilities.

For applications of this type, PACE uses a combination of the following rules to explore the state space:

- **R3**: *Generate states that result when a particular relaxation is applied on the master.*
- **R4**: *Generate states that result when a particular relaxation is applied on any one slave.*
- **R5**: *Generate states that result when a particular relaxation is applied on all nodes at the same time.*

In Redis, we use $R3$ and $R4$ but *not* $R5$: we use $R3$ to impose APM relaxations only on the master because the cluster can become unavailable for writes if only the master fails; we use $R4$ as reads can go to slaves. Similarly, in Kafka, we use $R3$ and $R5$ and *not* $R4$: we do not use $R4$ because all reads and writes go only through the leader; we use $R5$ to test states where the entire cluster can become unavailable because the cluster will be usable even if one node functions. MongoDB can be configured in many ways. We configure it much like an RSM system where it requires a majority for leader election and writes; hence, we use $R1$ and $R2$.

Examining a new distributed system with PACE requires developers to only understand whether the system implements a replicated state machine or not and how the master election works. Once this is known, PACE can be easily configured with the appropriate set of pruning rules. We believe that PACE can be readily helpful to developers given that they already know their system's protocols. We reiterate that the pruning rules do not guarantee finding all vulnerabilities; rather, they provide a set of guidelines to quickly search for problems. In the worst case, if no properties are known about a protocol, PACE can work in brute-force mode to find vulnerabilities.

3.3.3 Effectiveness of Pruning

To demonstrate the effectiveness of our pruning rules, we explored crash states of Redis and LogCabin with PACE and the brute-force approach. In Redis, for a simple workload on a three node cluster, brute-force needs to check 11,351 states whereas PACE only needs to check 1009 states. While exploring 11× fewer states, PACE found the same three vulnerabilities as the brute-force approach. In LogCabin, PACE discovers two vulnerabilities, checking 27,713 states in eight hours; the brute-force approach did not find any new vulnerabilities after running for over a week and exploring nearly 900,000 states. The reduction would be more pronounced as the number of nodes in a system increases.

3.4 Limitations and Caveats

PACE is *not complete* – it can miss vulnerabilities. Specifically, PACE exercises only one and the same reordering at a time across the set of nodes. For instance, consider two reorderings r_i and r_j. It is possible that no vulnerability is seen if r_i or r_j is applied individually on two nodes. But when r_i is applied on one node and r_j on the other, then it may lead to a vulnerability. PACE would miss such vulnerabilities. Note that if r_i and r_j can both individually cause a vulnerability, then PACE would catch both of them individually. This is a limitation in implementation and not a fundamental one. There is *no* similar limitation with partially persisting operations (i.e., PACE can partially persist different operations across nodes). Also, PACE does not focus on finding bugs in agreement protocols. We expand more on this topic later (§5).

System	Configuration	Workload	Checker
Redis	*appendfsync=always*, *min-slaves-to-write=2* and *wait*	update existing	old and new data (master and slave), *check-aof*, *check-dump*
ZooKeeper	Default	update existing	old and new data
LogCabin	Default	update existing	old and new data
etcd	Default	update existing	old and new data
RethinkDB	*durability=hard*, *writeack=majority*	update existing, insert new	old and new data
MongoDB	*W=3*, *journal=true*	update existing	old and new data
iNexus	Default	update existing, insert new	old and new data
Kafka	*flush.interval.msgs=1*, *min in-sync replicas=3*, *DirtyElection=False*	create topic, insert message	topic and message

Table 1: **Configurations, Workloads, and Checkers.** *The table shows the configuration, workloads and checkers for each system. We configured all systems with three nodes. The configuration settings ensure data is synchronously replicated and flushed to disk.*

4 Application Vulnerabilities Study

We studied eight widely used distributed systems spanning different domains including database caches (Redis v3.0.4), configuration stores (ZooKeeper v3.4.8, LogCabin v1.0.0, etcd v2.3.0), real-time databases (RethinkDB v2.2.5), document stores (MongoDB v3.0.11), key-value stores (iNexus v0.13), and message queues (Kafka v0.9.0). We tested MongoDB with two storage engines: WiredTiger [64] (MongoDB-WT) and RocksDB [86] (MongoDB-R). PACE found **26** unique vulnerabilities across the eight systems.

We first describe the workloads and checkers we used to detect vulnerabilities (§4.1). We then present a few example protocols and vulnerabilities to give an intuition of our methodology and the types of vulnerabilities discovered (§4.3). We then answer three important questions: Are there common patterns in file-system requirements (§4.4)? What are the consequences of the vulnerabilities discovered by PACE (§4.5)? How many vulnerabilities are exposed on real file systems (§4.6)? We then describe our experience with reporting the vulnerabilities to application developers (§4.7). We finally conclude by discussing the implications of our findings and the difficulties in fixing the discovered vulnerabilities (§4.8).

4.1 Application Workloads and Checkers

Most systems have configuration options that change user-level guarantees. We configured each system to provide the highest level of safety guarantees possible. When guarantees provided are unclear, our checkers check for typical user expectations; for example, data acknowledged as committed should not be lost in any case or the cluster should be available after recovering from crashes. Even though some applications do not explicitly guarantee such properties, we believe it is reasonable to test for such common expectations.

To test a system, we first construct a workload. Our workloads are not specifically crafted to expose vulnera-

```
## Workload ##
# Start cluster
# Insert new data
zk = client(hosts=server_ips)
zk.set("/mykey", "newvalue")
pace.acknowledged = True
# Stop cluster
```

```
## Checker ##
# Start cluster
# Check for data
retry_policy = retry(max_tries = r, delay = d,
backoff = b)
zk = client(hosts=server_ips, retry_policy)
ret, stat = zk.get("/mykey")
if request succeeded:
  if pace.acknowledged and ret == None:
    return 'data loss new commit'
  if pace.acknowledged and ret != 'newvalue':
    return 'corrupt'
  if not pace.acknowledged and ret == None:
    return 'data loss old commit'
else:
  return 'unavailable'
return 'correct'
# Stop cluster
```

Listing 1: **Workload and Checker.** *Simplified workload and checker for ZooKeeper.*

bilities, but rather are very natural and simple. Our workloads insert new data or update existing data and record the acknowledgment from the cluster. They are usually about 30-40 LOC.

To check each crash state, we implement a checker. The checker is conceptually simple; it starts the cluster with the crash state produced by PACE and checks for correctness by reading the data updated by the workload. If the data is lost, corrupted, or not retrievable, the checker flags the crash state incorrect. Further, our checkers invoke recovery tools mentioned in applications' documentation if an undesired output is observed. If the problem is fixed after invoking the recovery tool, then it is *not* reported as a vulnerability. Our checkers are about 100 LOC. Table 1 shows the configurations (that achieve the strongest safety guarantees), workloads, and checkers for all systems. Listing 1 shows the simplified pseudocode of the workload and the checker for ZooKeeper.

4.2 Vulnerability Accounting

A system has a crash vulnerability if a crash exposes a user-level guarantee violation. Counting such vulnerable places in the code is simple for single-machine applications. In a distributed system, multiple copies of the same code execute and so PACE needs to be careful in how it counts *unique* vulnerabilities.

We count only unique combinations of states that expose a vulnerability. Consider a sequence *S1* that creates (C), appends (A), and renames (R) a file. Assume that a node will not start if it crashes after C but before R. Assume there are three nodes in an RSM system and two crash after C but before R. In this case, the cluster can become unusable in four ways (C-C, CA-CA, C-CA, CA-C). We count all such instances as one vulnerability. If the third node crashes within this sequence, it will also be mapped onto the same vulnerability. If there is another different sequence *S2* that causes problems, a vulnerability could be exposed in many different ways as one node can crash within *S1* and another within *S2*. We associate all such combinations to two unique vulnerabilities, attributing to the atomicity of *S1* and *S2*.

PACE also associates each vulnerability with the application source code line using the stack trace information obtained during tracing. When many vulnerabilities map to the same source line, PACE considers that a single vulnerability. When we are unable to find the exact source lines for two different vulnerabilities, we count them as one. We note that our way of counting vulnerabilities results in a conservative estimate.

4.3 Example Protocols and Vulnerabilities

Figure 4 shows protocols and vulnerabilities in ZooKeeper, etcd, Redis, and Kafka. Due to space constraints, we show protocols only for four systems; protocol diagrams for other systems are publicly available [2].

RSM systems where vulnerabilities are exposed when APM relaxations are applied on a majority of nodes are represented using a grid. Figure 4(a) and 4(b) show the combinations of persistent states across two nodes in a three node ZooKeeper and etcd cluster, respectively. Operations that change persistent state are shown on the left (for one node) and the top (for the other node). A box *(i,j)* corresponds to a crash point where the first node crashes at operation *i* and the second at *j*. At each such crash point, PACE reorders other operations, or partially persists operations or both. A grey box denotes that the distributed execution did not reach that combination of states. A white box means that after applying all APM relaxations, PACE was unable to find a vulnerability. A black box denotes that when a specific relaxation (shown on the left) is applied, a vulnerability is exposed.

As shown in Figure 4(a), to maintain proposal information, ZooKeeper appends epoch numbers to temporary files, and renames them. If the renames are not atomic or reordered after a later write, the cluster becomes unavailable. If a log file creation and a subsequent append of header metadata are not atomically persisted, then the nodes fail to start. Similarly, the immediate truncate after log creation has to be atomically persisted for correct startup. Writes and appends during transactions, if reordered, can also cause node startup failures. ZooKeeper can lose data as it does not `fsync` the parent directory when a log is created.

Figure 4(b) shows the protocol and vulnerabilities in etcd. etcd creates a temporary write-ahead log (WAL),

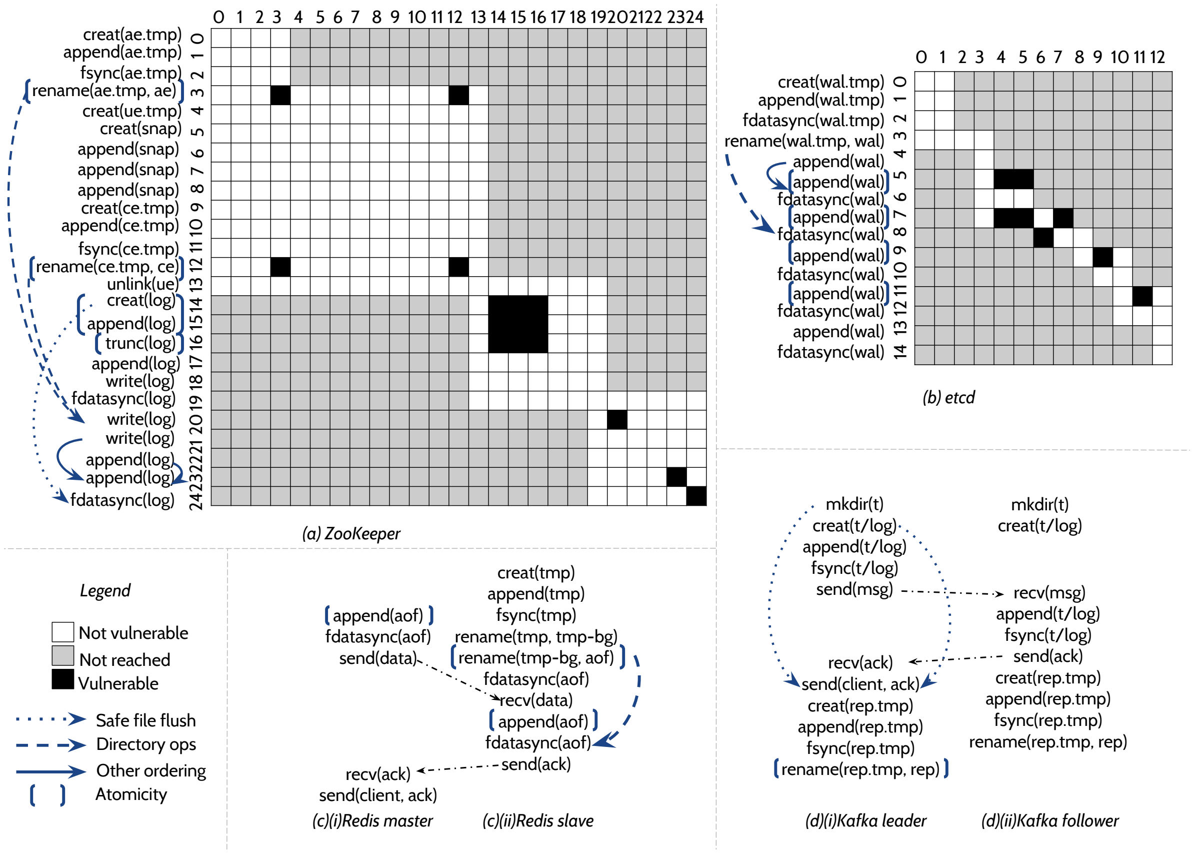


Figure 4: **Protocols and Vulnerabilities.** *(a), (b), (c), and (d) show protocols and vulnerabilities in ZooKeeper, etcd, Redis, and Kafka, respectively. States that are not vulnerable, that were not reached in the execution, and that are vulnerable are shown by white, grey, and black boxes, respectively. The annotations show how a particular state becomes vulnerable. In Zookeeper, box (24, 24) is vulnerable because both nodes crash after the final fdatasync but before the log creation is persisted. Atomicity vulnerabilities are shown with brackets enclosing the operations that need to be persisted atomically. The arrows show the ordering dependencies in the application protocol; if not satisfied, vulnerabilities are observed. Dotted, dashed, and solid arrows represent safe file flush, directory operation, and other ordering dependencies, respectively.*

appends some metadata, and renames it to create the final WAL. The WAL is appended, flushed, and then the client is acknowledged. We find that etcd cluster becomes unavailable if crashes occur when the WAL is appended; the nodes fail to start if the appends to the WAL are reordered or not persisted atomically. Also, if the rename of the WAL is reordered, a global data loss is observed.

Non-RSM systems where vulnerabilities are exposed even when relaxations are applied on a single machine are shown using trace pairs. As shown in Figure 4(c), Redis uses an append-only file to store user data. The master appends to the file and sends the update to slaves. Slaves, on startup, rewrite and rename the append-only file. When the master sends new data, the slaves append it to their append-only file and sync it. After the slaves respond, the client is acknowledged. Data loss windows are seen if the rename of the append-only file is not atomic or reordered after the final fdatasync. When the append is not atomic on the master, a user-visible silent corruption is observed. Moreover, the corrupted data is propagated from the master to the slaves, overriding their correct data. The same append (which maps to the same source line) on the slave results in a window of silent corruption. The window closes eventually since the slaves sync the data from the master on startup.

Figure 4(d) shows the update protocol of Kafka. Kafka creates a log file in the topic directory to store messages. When a message is added, the leader appends the message and flushes the log. It then contacts the followers which perform the same operation and respond. After acknowledging the client, the replication offset (that tracks which messages are replicated to other brokers) is appended to a temporary file, flushed, and renamed to the replication-offset-checkpoint file. The log can be lost after a crash because its parent directory is not flushed after the log creation. If the log is lost on the master, then the data is globally lost since the master instructs the slaves also to drop the messages in the log. Similarly, Kafka can lose a message topic altogether since the parent directory of the topic directory is not explicitly flushed.

We observe that some systems (e.g., Redis, Kafka) do not effectively use redundancy as a source of recovery.

	FS Requirements							Failure Consequences								
		Atomic-ity			Order-ing				Data loss		Un-avail-able		Window			
System	**Inter-syscall atomicity**	Appends and truncates	Rename (dest link absent)	Rename (dest link exists)	Safe file flush	Directory ops	Other	**Silent corruption**	Old commit	New commit	Metadata corruption	User data corruption	Corruption	Data loss old commit	Data loss new commit	**Unique vulnerabilities**
Redis		1		1		1		1					1	1	1	**3**
ZooKeeper	1	1		1	1	1	1			1	4	1				**6**
LogCabin	1		1						1		1			1		**2**
etcd		1				1	1			1	1	2				**3**
RethinkDB																
MongoDB-WT				1							1					**1**
MongoDB-R		1		1	1	1	1			3	3					**5**
iNexus				1		1	1		1	1	2					**3**
Kafka				1	2					3						**3**
Total	2	4	1	6	4	5	4	1	2	9	12	3	1	2	1	**26**

Table 2: **Vulnerabilities - Types and Consequences.** *The table shows the unique vulnerabilities categorized by file-system requirements and consequences.*

For instance, in these systems, a local problem (such as a local corruption or data loss) which results due to a relaxation on a single node, can easily become a global vulnerability such as a user-visible silent corruption or data loss. In such situations, these systems miss opportunities to use other intact replicas to recover from the local problem. Moreover, such local problems are propagated to other intact replicas, overriding their correct data.

4.4 Patterns in File-system Requirements

Table 2 shows file-system requirements across systems. We group the results into three patterns:

Inter-Syscall Atomicity. ZooKeeper and LogCabin require inter system call atomicity (multiple system calls need to be atomically persisted). In both these systems, when a new log file is initialized, the creat and the initial append of the log header need to be atomically persisted. If the log initialization is partially persisted, the cluster becomes unavailable. Vulnerabilities due to inter system call atomicity requirements can occur on all file systems irrespective of how they persist operations.

Atomicity within System calls. We find that seven applications require system calls to be atomically persisted. Eleven unique vulnerabilities are observed when system calls are not persisted atomically. Six out of the eleven vulnerabilities are dependent on atomic replace by rename (destination link already exists), one on atomic create by rename (destination link does not exist), and four on atomic truncates or appends. Four applications require appends or truncates to be atomic. Redis, ZooKeeper, and etcd can handle appended portions filled with zeros but not garbage.

Ordering between System calls. Six applications expect system calls to be persisted in order. Kafka and ZooKeeper suffer from data loss since they expect the safe file flush property from the file system. To persist a file's directory entry, the parent directory has to be explicitly flushed to avoid such vulnerabilities. We found that reordering directory operations can cause vulnerabilities. We found that five applications depend on ordered renames: Redis exhibits a data loss window, etcd permanently loses data, ZooKeeper, MongoDB-R, and iNexus fail to start. Four applications require other operations (appends and writes) to be ordered for correct behavior.

4.5 Vulnerability Consequences

Table 2 shows the vulnerability consequences. We find that all vulnerabilities have severe consequences like silent corruption, data loss, or cluster unavailability. Redis silently returns and propagates corrupted data from the master to slaves even if slaves have correct older version of data. Redis also has a silent corruption window when reads are performed on slaves. While only one system silently corrupts and propagates corrupted data, six out of eight systems are affected by permanent data loss. Depending on the crash state, previously committed data can be lost when new data is inserted or the newly inserted data can be lost after acknowledgment. Redis exhibits a data loss window that is exposed when reads are performed on the slaves. As slaves continuously sync data from the master, the window eventually closes.

Cluster unavailability occurs when nodes fail to start due to corrupted application data or metadata. ZooKeeper and etcd fail to start if CRC checksums mismatch in user data. MongoDB-WT fails to start if the *turtle* file is missing and MongoDB-R fails to start if the *sstable* file is missing or there is a mismatch in the *current* and *manifest* files. LogCabin and iNexus skip log entries when checksums do not match but fail to start if metadata is corrupted. LogCabin fails to start when an unexpected segment metadata version is found. Similarly, ZooKeeper fails to start on unexpected epoch values. While some of these scenarios can be fixed by expert application users, the process is intricate and error prone.

We note that the vulnerabilities are specific to our simple workloads and all vulnerabilities reported by PACE have harmful consequences. More complex workloads and checkers that assert more subtle invariants are bound to find more vulnerabilities.

4.6 Impact on Real File Systems

We configured PACE with APMs of real file systems. Table 3 shows the vulnerabilities on each file system. We observe that many vulnerabilities can occur on all examined file systems. Only two vulnerabilities are observed in ext3-j (data-journaling) as all operations are persisted in order. All vulnerabilities that occur on our default

	ext2	ext3-w	ext3-o	ext4-o	ext3-j	btrfs
Redis	3	1				1
ZooKeeper	6	3	1	1	1	3
LogCabin	2	1	1	1	1	1
etcd	3	2				
MongoDB-WT	1					
MongoDB-R	5	2	2	2		3
iNexus	2		1	1		2
Kafka	3					
Total	26	9	5	5	2	10

Table 3: **Vulnerabilities on Real File Systems.** *The table shows the number of vulnerabilities on commonly used file systems.*

APM are also exposed on ext2. Applications are vulnerable even on Linux's default file system (ext4 ordered mode). Many of the vulnerabilities are exposed on btrfs as it reorders directory operations. In summary, the vulnerabilities are exposed on many current file systems on which distributed storage systems run today.

4.7 Confirmation of Problems Found

We reported 18 of the discovered vulnerabilities to application developers. We confirmed that the reported issues cause serious problems (such as data loss and unavailability) to users of the system. Seven out of the 18 reported issues were assigned to developers and fixed [32–34, 56, 87, 88]. Another five issues have been acknowledged or assigned to developers. Out of this five, two in Kafka were already known [48]. Other issues are still open and under consideration. We found that distributed storage system developers, in general, are responsive to such bug reports for two reasons. First, we believe developers consider crashes very important in distributed systems compared to single-machine applications. Second, the discovered vulnerabilities due to crashes affect their users directly (for example, data loss and cluster unavailability).

We found that users and random-crash testing have also occasionally encountered the same vulnerabilities that were systematically discovered by PACE. However, PACE diagnoses the underlying root cause and provides information of the problematic source code line, easing the process of fixing these vulnerabilities.

4.8 Discussion

We first discuss the immediate implications of our findings in building distributed storage systems atop file systems. Next, we discuss the difficulties in fixing some of the discovered vulnerabilities.

4.8.1 Implications

We find that redundancy by replication is not the panacea for constructing reliable storage systems. Although replication can help with single node failures, correlated crashes still remain a problem. We find that application protocols, when driven to corner cases, can often override correct versions of data with corrupted data or older versions without considering how the system reached such a state. For example, Redis and Kafka can propagate corrupted data and data loss to slaves, respectively. Similarly, RSM systems override correct newer versions of data on other nodes when a majority of nodes have lost the data; a better recovery strategy could use the unaffected replicas to fix the loss of acknowledged data even if the data is lost on a majority of nodes. We believe replication protocols and local storage protocols should be designed in tandem to avoid such undesired behaviors.

System designers need to be careful about two problems when embracing *layered* software. First, the reliability of the entire system depends on individual components. MongoDB's reliability varies depending on the storage engine (WiredTiger or RocksDB). Second, separate well-tested components when integrated can bring unexpected problems. In the version of MongoDB we tested, we found that correct options are not passed from upper layers to RocksDB, resulting in a data loss. Similarly, iNexus uses a modified version of LevelDB which does not flush writes to disk when transactions commit. Applications need to clearly understand the guarantees provided by components when using them.

We find that a few applications are overly cautious in how they update file-system state. LogCabin flushes files and directories after every operation. Though this avoids many reordering vulnerabilities, it does not fix atomicity vulnerabilities. Issuing `fsync` at various places does not completely avoid reliability problems. Also, the implication of too much caution is clear: low performance. While this approach is reasonable for configuration stores, key-value stores need a better way to achieve the same effect without compromising performance.

All modern distributed storage system run on top of a variety of file systems that provide different crash guarantees. We advocate that distributed storage systems should understand and document on which file systems their protocols work correctly to help practitioners make conscious deployment decisions.

4.8.2 Difficulties in Fixing

Now we discuss the difficulties in fixing the discovered vulnerabilities. The effort to fix the vulnerabilities varies significantly. While some of them are simple implementation-level fixes, many of them are fundamental problems that require rethinking how distributed crash recovery protocols are designed.

Some vulnerabilities (such as those due to non-atomic renames) are automatically masked in modern file systems; these possess a practical concern only when the applications are run on file systems that do not provide such guarantees (e.g., ext2). While only some vulnerabilities can be easily fixed, many vulnerabilities are fundamentally hard to fix and they fall into three categories.

First, some vulnerabilities cannot be fixed by current file system interfaces or straightforward changes in application local update protocols. For example, consider the inter-syscall atomicity vulnerabilities and the non-atomic multi-block appends and truncates. These vulnerabilities are exposed on all current file systems since `POSIX` does not provide a way to atomically persist multiple system calls or a write that spans multiple blocks. While a different interface that allows multiple system calls to be atomically persisted can help, such an interface is far from reality in current commodity file systems.

Second, many reordering vulnerabilities can be fixed by carefully issuing `fsync` at correct places in the local update protocol. However, applications may not be willing to do so, given the clear performance impact in the common case update protocol code.

Third, sometimes the programming environment may constrain applications from utilizing some file system interfaces, leading to vulnerabilities. For example, consider the safe file flush and directory operation reordering vulnerabilities in ZooKeeper and Kafka. These vulnerabilities arise because these systems are written in Java in which `fsync` cannot be readily issued on directories.

In all the above cases, simply fixing the local update protocols is not a feasible solution. Fixing these fundamental problems requires carefully designing local and global recovery protocols that interact correctly to fix the problem using other intact replicas.

5 Related Work

Recent work has demonstrated that file-system behaviors vary widely [4, 14, 70–73]. We derive our APM specifications from our previous work [70]. Our previous work also developed Alice to uncover single-machine crash vulnerabilities. Tools like Alice cannot be directly applied to distributed systems as they do not track cross node dependencies. If applied, such tools may report spurious vulnerabilities. Although such tools can be applied in stand-alone mode like in ZooKeeper [99], many code paths would not be exercised and thus miss important vulnerabilities. Zheng et al. [98] find crash vulnerabilities in databases. Unlike our work, Zheng et al. do not systematically explore all states that can occur in an execution. They find vulnerabilities that can commonly occur: they do not model file-system behavior closely and therefore cannot explore all corner cases.

PACE is complementary to distributed model checkers [29, 40, 42, 54, 95, 96]: bugs due to file-system behaviors discovered by PACE cannot be discovered by existing model checkers and bugs due to network message re-orderings cannot be discovered by PACE. Distributed model checkers use dynamic partial-order based techniques to reduce state space explosion. SAMC [54] can also induce crashes and reboots in addition to reordering messages. To reduce state space, SAMC uses semantic information which requires testers to write protocol-specific rules for a target system. PACE uses only high-level protocol-awareness to prune the state space and does not require any code as input. Jepsen [50] is a tool that tests distributed systems under faulty networks and partial failures. Similar to distributed model checkers, such tools are complementary to PACE.

Previous tools focus solely on either single-node file system behavior or distributed consensus and thus cannot understand the interaction of distributed recovery protocol and a local-storage protocol. To our knowledge, our work is the first to consider file-system behaviors in the context of distributed storage systems. It is difficult for other distributed model checkers to reproduce the vulnerabilities found by PACE because they run the system on top of already implemented storage stacks. PACE models the file system used by the distributed system and thus can check how a distributed storage system will work on any current or future file system.

6 Conclusion

Modern distributed storage systems suffer from correlated crash vulnerabilities and subtleties in local file-system behavior influence the correctness of distributed update protocols. We present PACE a tool that can effectively search for correlated crash vulnerabilities by pruning the search space. We study eight popular distributed storage systems using PACE and find 26 unique vulnerabilities. As modern distributed storage systems are becoming the primary choice for storing and managing critical user data, tools such as PACE are increasingly vital to uncover reliability problems. Source code of PACE, workloads, checkers, and details of the discovered vulnerabilities are publicly available at `http://research.cs.wisc.edu/adsl/Software/pace/`.

Acknowledgments

We thank the anonymous reviewers and Jeffrey Chase (our shepherd) for their insightful comments. We thank the members of the ADSL for their valuable discussions. This material was supported by funding from NSF grants CNS-1419199, CNS-1421033, CNS-1319405, and CNS-1218405, DOE grant DE-SC0014935, as well as donations from EMC, Facebook, Google, Huawei, Microsoft, NetApp, Samsung, Seagate, Veritas, and VMware. Finally, we thank CloudLab [85] for providing a great computing environment for running our experiments. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and may not reflect the views of NSF, DOE, or other institutions.

References

[1] Necessary step(s) to synchronize filename operations on disk. http://austingroupbugs.net/view.php?id=672.

[2] Pace Tool and Results. http://research.cs.wisc.edu/adsl/Software/pace/.

[3] Atul Adya, William J. Bolosky, Miguel Castro, Gerald Cermak, Ronnie Chaiken, John R. Douceur, Jon Howell, Jacob R. Lorch, Marvin Theimer, and Roger P. Wattenhofer. Farsite: Federated, Available, and Reliable Storage for an Incompletely Trusted Environment. *SIGOPS Oper. Syst. Rev.*, 36(SI):1–14, December 2002.

[4] Ramnatthan Alagappan, Vijay Chidambaram, Thanumalayan Sankaranarayana Pillai, Aws Albarghouthi, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. Beyond Storage APIs: Provable Semantics for Storage Stacks. In *Proceedings of the 15th USENIX Conference on Hot Topics in Operating Systems*, HOTOS'15, pages 20–20, Berkeley, CA, USA, 2015. USENIX Association.

[5] Apache. Apache ZooKeeper. https://zookeeper.apache.org/.

[6] Apache. Kafka. http://kafka.apache.org/.

[7] Apache Cassandra. Cassandra Replication. http://docs.datastax.com/en/cassandra/2.0/cassandra/architecture/architectureDataDistributeReplication_c.html.

[8] ArchLinux. f2fs. https://wiki.archlinux.org/index.php/F2FS.

[9] Remzi H. Arpaci-Dusseau and Andrea C. Arpaci-Dusseau. *Operating Systems: Three Easy Pieces*. Arpaci-Dusseau Books, 0.91 edition, May 2015.

[10] Özalp Babaoğlu and Keith Marzullo. Distributed Systems (2Nd Ed.). pages 55–96, 1993.

[11] Mehmet Bakkaloglu, Jay J Wylie, Chenxi Wang, and Gregory R Ganger. On Correlated Failures in Survivable Storage Systems . Technical report, DTIC Document, 2002.

[12] Kenneth P. Birman and Thomas A. Joseph. Reliable Communication in the Presence of Failures. *ACM Trans. Comput. Syst.*, 5(1):47–76, January 1987.

[13] Andrew D. Birrell, Roy Levin, Michael D. Schroeder, and Roger M. Needham. Grapevine: An Exercise in Distributed Computing. *Commun. ACM*, 25(4):260–274, April 1982.

[14] James Bornholt, Antoine Kaufmann, Jialin Li, Arvind Krishnamurthy, Emina Torlak, and Xi Wang. Specifying and Checking File System Crash-Consistency Models. In *Proceedings of the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '16, pages 83–98, New York, NY, USA, 2016. ACM.

[15] Mike Burrows. The Chubby Lock Service for Loosely-coupled Distributed Systems. In *Proceedings of the 7th Symposium on Operating Systems Design and Implementation*, OSDI '06, pages 335–350, Berkeley, CA, USA, 2006. USENIX Association.

[16] Marco Canini, Vojin Jovanovic, Daniele Venzano, Gautam Kumar, Dejan Novakovic, and Dejan Kostic. Checking for Insidious Faults in Deployed Federated and Heterogeneous Distributed Systems. Technical report, 2011.

[17] Cassandra. Apache Cassandra. https://academy.datastax.com/resources/brief-introduction-apache-cassandra.

[18] K. Mani Chandy and Leslie Lamport. Distributed Snapshots: Determining Global States of Distributed Systems. *ACM Trans. Comput. Syst.*, 3(1):63–75, February 1985.

[19] Fay Chang, Jeffrey Dean, Sanjay Ghemawat, Wilson C. Hsieh, Deborah A. Wallach, Mike Burrows, Tushar Chandra, Andrew Fikes, and Robert E. Gruber. Bigtable: A Distributed Storage System for Structured Data. pages 15–15, 2006.

[20] Asaf Cidon, Stephen M. Rumble, Ryan Stutsman, Sachin Katti, John Ousterhout, and Mendel Rosenblum. Copysets: Reducing the Frequency of Data Loss in Cloud Storage. In *Proceedings of the 2013 USENIX Conference on Annual Technical Conference*, USENIX ATC'13, pages 37–48, Berkeley, CA, USA, 2013. USENIX Association.

[21] ComputerWorldUK. Lightning strikes Amazon and Microsoft data centres. http://www.computerworlduk.com/galleries/infrastructure/ten-datacentre-disasters-that-brought-firms-offline-3593580/#5.

[22] Brian F. Cooper, Raghu Ramakrishnan, Utkarsh Srivastava, Adam Silberstein, Philip Bohannon, Hans-Arno Jacobsen, Nick Puz, Daniel Weaver, and Ramana Yerneni. PNUTS: Yahoo!'s Hosted Data Serving Platform. *Proc. VLDB Endow.*, 1(2):1277–1288, August 2008.

[23] CoreOS. etcd. https://coreos.com/etcd/.

[24] Flaviu Cristian, Houtan Aghili, Raymond Strong, and Danny Dolev. *Atomic Broadcast: From Simple Message Diffusion to Byzantine Agreement*. Citeseer, 1986.

[25] DataCenterDynamics. Lessons from the Singapore Exchange failure. http://www.datacenterdynamics.com/power-cooling/lessons-from-the-singapore-exchange-failure/94438.fullarticle.

[26] DataCenterKnowledge. Lightning Disrupts Google Cloud Services. http://www.datacenterknowledge.com/archives/2015/08/19/lightning-strikes-google-data-center-disrupts-cloud-services/.

[27] Datastax. Netflix Cassandra use case. http://www.datastax.com/resources/casestudies/netflix.

[28] Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshall, and Werner Vogels. Dynamo: Amazon's Highly Available Key-value Store. volume 41, pages 205–220, New York, NY, USA, October 2007. ACM.

[29] Pantazis Deligiannis, Matt McCutchen, Paul Thomson, Shuo Chen, Alastair F. Donaldson, John Erickson, Cheng Huang, Akash Lal, Rashmi Mudduluru, Shaz Qadeer, and Wolfram Schulte. Uncovering Bugs in Distributed Storage Systems during Testing (Not in Production!). In *14th USENIX Conference on File and Storage Technologies (FAST 16)*, pages 249–262, Santa Clara, CA, Feb 2016. USENIX Association.

[30] Docker. Docker. https://www.docker.com/.

[31] Robert Escriva, Bernard Wong, and Emin Gün Sirer. HyperDex: A Distributed, Searchable Key-value Store. In *Proceedings of the ACM SIGCOMM 2012 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication*, SIGCOMM '12, pages 25–36, New York, NY, USA, 2012. ACM.

[32] etcd. Possible cluster unavailability on few file systems. https://github.com/coreos/etcd/issues/6379.

[33] etcd. Possible cluster unavailbility. https://github.com/coreos/etcd/issues/6378.

[34] etcd. Possible data loss – fsync parent directories. https://github.com/coreos/etcd/issues/6378.

[35] Daniel Ford, François Labelle, Florentina I. Popovici, Murray Stokely, Van-Anh Truong, Luiz Barroso, Carrie Grimes, and Sean Quinlan. Availability in Globally Distributed Storage Systems. In *Presented as part of the 9th USENIX Symposium on Operating Systems Design and Implementation*, Berkeley, CA, 2010. USENIX.

[36] Dennis Geels, Gautam Altekar, Petros Maniatis, Timothy Roscoe, and Ion Stoica. Friday: Global Comprehension for Distributed Replay. In *Proceedings of the 4th USENIX Conference on Networked Systems Design &#38; Implementation*, NSDI'07, pages 21–21, Berkeley, CA, USA, 2007. USENIX Association.

[37] Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung. The Google File System. In *Proceedings of the Nineteenth ACM Symposium on Operating Systems Principles*, SOSP '03, pages 29–43, New York, NY, USA, 2003. ACM.

[38] Google. Google Cloud Status. https://status.cloud.google.com/incident/compute/15056#5719570367119360.

[39] Google Code University. Introduction to Distributed System Design. http://www.hpcs.cs.tsukuba.ac.jp/~tatebe/lecture/h23/dsys/dsd-tutorial.html.

[40] Rachid Guerraoui and Maysam Yabandeh. Model Checking a Networked System Without the Network. In *Proceedings of the 8th USENIX Conference on Networked Systems Design and Implementation*, NSDI'11, pages 225–238, Berkeley, CA, USA, 2011. USENIX Association.

[41] Haryadi S. Gunawi, Vijayan Prabhakaran, Swetha Krishnan, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. Improving File System Reliability with I/O Shepherding. In *Proceedings of the 21st ACM Symposium on Operating Systems Principles 2007, SOSP 2007, Stevenson, Washington, USA, October 14-17, 2007*, pages 293–306, 2007.

[42] Huayang Guo, Ming Wu, Lidong Zhou, Gang Hu, Junfeng Yang, and Lintao Zhang. Practical Software Model Checking via Dynamic Interface Reduction. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, SOSP '11, pages 265–278, New York, NY, USA, 2011. ACM.

[43] Andreas Haeberlen, Alan Mislove, and Peter Druschel. Glacier: Highly Durable, Decentralized Storage Despite Massive Correlated Failures. In *Proceedings of the 2Nd Conference on Symposium on Networked Systems Design & Implementation - Volume 2*, NSDI'05, pages 143–158, Berkeley, CA, USA, 2005. USENIX Association.

[44] Chris Hawblitzel, Jon Howell, Manos Kapritsos, Jacob R. Lorch, Bryan Parno, Michael L. Roberts, Srinath Setty, and Brian Zill. IronFleet: Proving Practical Distributed Systems Correct. In *Proceedings of the 25th Symposium on Operating Systems Principles*, SOSP '15, pages 1–17, New York, NY, USA, 2015. ACM.

[45] Patrick Hunt, Mahadev Konar, Flavio P. Junqueira, and Benjamin Reed. ZooKeeper: Wait-free Coordination for Internet-scale Systems. In *Proceedings of the 2010 USENIX Conference on USENIX Annual Technical Conference*, USENIXATC'10, pages 11–11, Berkeley, CA, USA, 2010. USENIX Association.

[46] iNexus. iNexus. `https://github.com/baidu/ins`.

[47] Kafka. Kafka Disks and Filesystem. `https://kafka.apache.org/081/ops.html`.

[48] Kafka. Possible data loss. `https://issues.apache.org/jira/browse/KAFKA-4127`.

[49] Kimberley Keeton, Cipriano Santos, Dirk Beyer, Jeffrey Chase, and John Wilkes. Designing for Disasters. In *Proceedings of the 3rd USENIX Conference on File and Storage Technologies*, FAST '04, pages 59–62, Berkeley, CA, USA, 2004. USENIX Association.

[50] Kyle Kingsbury. Jepsen. `http://jepsen.io/`.

[51] Leslie Lamport. Paxos Made Simple. *ACM Sigact News*, 32(4):18–25, 2001.

[52] Leslie Lamport, Robert Shostak, and Marshall Pease. The Byzantine Generals Problem. *ACM Trans. Program. Lang. Syst.*, 4(3):382–401, July 1982.

[53] Butler Lampson and Howard Sturgis. *Crash recovery in a distributed data storage system*. Xerox Palo Alto Research Center Palo Alto, California, 1979.

[54] Tanakorn Leesatapornwongsa, Mingzhe Hao, Pallavi Joshi, Jeffrey F. Lukman, and Haryadi S. Gunawi. SAMC: Semantic-aware Model Checking for Fast Discovery of Deep Bugs in Cloud Systems. In *Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation*, OSDI'14, pages 399–414, Berkeley, CA, USA, 2014. USENIX Association.

[55] Barbara Liskov and James Cowling. Viewstamped Replication Revisited. Technical Report MIT-CSAIL-TR-2012-021, MIT, July 2012.

[56] LogCabin. Cluster unavailable due to power failures. `https://github.com/logcabin/logcabin/issues/221`.

[57] LogCabin. LogCabin. `https://github.com/logcabin/logcabin`.

[58] Marshall Kirk McKusick and Jeffery Roberson. Journaled Soft-updates. *Proceedings of EuroBSDCon*, 2010.

[59] MongoDB. Introduction to MongoDB. `https://docs.mongodb.org/manual/introduction/`.

[60] MongoDB. MongoDB. `https://www.mongodb.org/`.

[61] MongoDB. MongoDB at ebay. `https://www.mongodb.com/presentations/mongodb-ebay`.

[62] MongoDB. MongoDB Platform Specific Considerations. `https://docs.mongodb.org/manual/administration/production-notes/#platform-specific-considerations`.

[63] MongoDB. MongoDB Replication. `https://docs.mongodb.org/manual/replication/`.

[64] MongoDB. MongoDB WiredTiger. `https://docs.mongodb.org/manual/core/wiredtiger/`.

[65] Suman Nath, Haifeng Yu, Phillip B. Gibbons, and Srinivasan Seshan. Subtleties in Tolerating Correlated Failures in Wide-area Storage Systems. In *Proceedings of the 3rd Conference on Networked Systems Design & Implementation - Volume 3*, NSDI'06, pages 17–17, Berkeley, CA, USA, 2006. USENIX Association.

[66] Rajesh Nishtala, Hans Fugal, Steven Grimm, Marc Kwiatkowski, Herman Lee, Harry C. Li, Ryan McElroy, Mike Paleczny, Daniel Peek, Paul Saab, David Stafford, Tony Tung, and Venkateshwaran Venkataramani. Scaling Memcache at Facebook. In *Proceedings of the 10th USENIX Conference on Networked Systems Design and Implementation*, nsdi'13, pages 385–398, Berkeley, CA, USA, 2013. USENIX Association.

[67] Brian M. Oki and Barbara H. Liskov. Viewstamped Replication: A New Primary Copy Method to Support Highly-Available Distributed Systems. In *Proceedings of the Seventh Annual ACM Symposium on Principles of Distributed Computing*, PODC '88, pages 8–17, New York, NY, USA, 1988. ACM.

[68] Diego Ongaro and John Ousterhout. In Search of an Understandable Consensus Algorithm. In *Proceedings of the 2014 USENIX Conference on USENIX Annual Technical Conference*, USENIX ATC'14, pages 305–320, Berkeley, CA, USA, 2014. USENIX Association.

[69] David A. Patterson, Garth Gibson, and Randy H. Katz. A Case for Redundant Arrays of Inexpensive Disks (RAID). In *Proceedings of the 1988 ACM SIGMOD International Conference on Management of Data*, SIGMOD '88, pages 109–116, New York, NY, USA, 1988. ACM.

[70] Thanumalayan Sankaranarayana Pillai, Vijay Chidambaram, Ramnatthan Alagappan, Samer Al-Kiswany, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. All File Systems Are Not Created Equal: On the Complexity of Crafting Crash-consistent Applications. In *Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation*, OSDI'14, pages 433–448, Berkeley, CA, USA, 2014. USENIX Association.

[71] Thanumalayan Sankaranarayana Pillai, Vijay Chidambaram, Ramnatthan Alagappan, Samer Al-Kiswany, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. Crash Consistency. *Communications of the ACM*, 58(10):46–51, October 2015.

[72] Thanumalayan Sankaranarayana Pillai, Vijay Chidambaram, Ramnatthan Alagappan, Samer Al-Kiswany, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. Crash Consistency: Rethinking the Fundamental Abstractions of the File System. *Communications of the ACM*, 13(7), July 2015.

[73] Thanumalayan Sankaranarayana Pillai, Vijay Chidambaram, Joo-Young Hwang, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. Towards Efficient, Portable Application-level Consistency. In *Proceedings of the 9th Workshop on Hot Topics in Dependable Systems*, HotDep '13, pages 8:1–8:6, New York, NY, USA, 2013. ACM.

[74] Redis. Instagram Architecture. `http://highscalability.com/blog/2012/4/9/the-instagram-architecture-facebook-bought-for-a-cool-billio.html`.

[75] Redis. Introduction to Redis. `http://redis.io/topics/introduction`.

[76] Redis. Redis. `http://redis.io/`.

[77] Redis. Redis at Flickr. `http://code.flickr.net/2014/07/31/redis-sentinel-at-flickr/`.

[78] Redis. Redis at GitHub. `http://nosql.mypopescu.com/post/1164218362/redis-at-github`.

[79] Redis. Redis at Pinterest. `http://highscalability.com/blog/2012/4/9/the-instagram-architecture-facebook-bought-for-a-cool-billio.html`.

[80] Redis. Redis Replication. `http://redis.io/topics/replication`.

[81] Redis. Virtual Memory – Redis . `http://redis.io/topics/virtual-memory`.

[82] Redis. Who's using Redis? `http://redis.io/topics/whos-using-redis`.

[83] RethinkDB. RethinkDB. `https://www.rethinkdb.com/`.

[84] RethinkDB. RethinkDB Replication. `https://www.rethinkdb.com/docs/sharding-and-replication/`.

[85] Robert Ricci, Eric Eide, and The CloudLab Team. Introducing CloudLab: Scientific infrastructure for advancing cloud architectures and applications. *USENIX ;login:*, 39(6), December 2014.

[86] RocksDB. RocksDB. `http://rocksdb.org/blog/1967/integrating-rocksdb-with-mongodb-2/`.

[87] Mongo RocksDB. Data loss – fsync parent directory on file creation and rename. `https://github.com/mongodb-partners/mongo-rocks/issues/35`.

[88] Mongo RocksDB. Mongodb - rocksdb data loss bug. `https://groups.google.com/forum/#!topic/mongodb-dev/X9LQOorieas`.

[89] Yasushi Saito and Marc Shapiro. Optimistic Replication. *ACM Comput. Surv.*, 37(1):42–81, March 2005.

[90] Fred B. Schneider. Implementing Fault-tolerant Services Using the State Machine Approach: A Tutorial. *ACM Comput. Surv.*, 22(4):299–319, December 1990.

[91] Douglas B. Terry, Marvin Theimer, Karin Petersen, Alan J. Demers, Mike Spreitzer, and Carl Hauser. Managing Update Conflicts in Bayou, a Weakly Connected Replicated Storage System. In *Proceedings of the Fifteenth ACM Symposium on Operating System Principles, SOSP 1995, Copper Mountain Resort, Colorado, USA, December 3-6, 1995*, pages 172–183, 1995.

[92] TheRegister. Admin downs entire Joyent data center. `http://www.theregister.co.uk/2014/05/28/joyent_cloud_down/`.

[93] Twitter. Twitter Blogs. `https://blog.twitter.com/2015/handling-five-billion-sessions-a-day-in-real-time`.

[94] Maysam Yabandeh, Nikola Knežević, Dejan Kostić, and Viktor Kuncak. Predicting and Preventing Inconsistencies in Deployed Distributed Systems. *ACM Trans. Comput. Syst.*, 28(1):2:1–2:49, August 2010.

[95] Maysam Yabandeh and Dejan Kostić. DPOR-DS: Dynamic Partial Order Reduction in Distributed Systems. Technical report, 2009.

[96] Junfeng Yang, Tisheng Chen, Ming Wu, Zhilei Xu, Xuezheng Liu, Haoxiang Lin, Mao Yang, Fan Long, Lintao Zhang, and Lidong Zhou. MODIST: Transparent Model Checking of Unmodified Distributed Systems. In *Proceedings of the 6th USENIX Symposium on Networked Systems Design and Implementation*, NSDI'09, pages 213–228, Berkeley, CA, USA, 2009. USENIX Association.

[97] YCombinator. Joyent us-east-1 rebooted due to operator error. `https://news.ycombinator.com/item?id=7806972`.

[98] Mai Zheng, Joseph Tucek, Dachuan Huang, Feng Qin, Mark Lillibridge, Elizabeth S. Yang, Bill W. Zhao, and Shashank Singh. Torturing Databases for Fun and Profit. In *Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation*, OSDI'14, pages 449–464, Berkeley, CA, USA, 2014. USENIX Association.

[99] ZooKeeper. ZooKeeper Standalone Operation. `https://zookeeper.apache.org/doc/r3.3.3/zookeeperStarted.html#sc_InstallingSingleMode`.

Incremental Consistency Guarantees for Replicated Objects

Rachid Guerraoui, Matej Pavlovic, and Dragos-Adrian Seredinschi*

School of Computer and Communication Sciences,
École Polytechnique Fédérale de Lausanne (EPFL), Switzerland
{*rachid.guerraoui, matej.pavlovic, dragos-adrian.seredinschi*}*@epfl.ch*

Abstract

Programming with replicated objects is difficult. Developers must face the fundamental trade-off between consistency and performance head on, while struggling with the complexity of distributed storage stacks. We introduce *Correctables*, a novel abstraction that hides most of this complexity, allowing developers to focus on the task of balancing consistency and performance. To aid developers with this task, Correctables provide *incremental consistency guarantees*, which capture successive refinements on the result of an ongoing operation on a replicated object. In short, applications receive both a preliminary—fast, possibly inconsistent—result, as well as a final—consistent—result that arrives later.

We show how to leverage incremental consistency guarantees by speculating on preliminary values, trading throughput and bandwidth for improved latency. We experiment with two popular storage systems (Cassandra and ZooKeeper) and three applications: a Twissandra-based microblogging service, an ad serving system, and a ticket selling system. Our evaluation on the Amazon EC2 platform with YCSB workloads A, B, and C shows that we can reduce the latency of strongly consistent operations by up to 40% (from 100*ms* to 60*ms*) at little cost (10% bandwidth increase, 6% throughput drop) in the ad system. Even if the preliminary result is frequently inconsistent (25% of accesses), incremental consistency incurs a bandwidth overhead of only 27%.

1. Introduction

Replication is a crucial technique for achieving performance—i.e., high availability and low latency—in large-scale applications. Traditionally, strong consistency protocols hide replication and ensure correctness by exposing a single-copy abstraction over replicated objects [26, 46]. There is a trade-off, however, between consistency and performance [14, 21, 33]. Weak consistency [28] boosts performance, but introduces the possibility of incorrect (anomalous) behavior.

A common argument in favor of weak consistency is that such anomalous behavior is rare in practice. Indeed, studies reveal that on expectation, weakly consistent values are often correct even with respect to strong consistency [19, 55]. Applications which primarily demand performance thus forsake stronger models and resort to weak consistency [16, 28].

There are cases, however, where applications often diverge from correct behavior due to weak consistency. As an extreme example, an execution of YCSB workload A [25] in Cassandra [45] on a small $1K$ objects dataset can reveal stale values for 25% of weakly consistent read operations (Figure 7 in §6). This happens when using the *Latest* distribution, where read activity is skewed towards popular items [25]. In other cases, even very rare anomalies are unacceptable (e.g., when handling sensitive data such as user passwords), making strongly consistent access a necessity. For this class of applications, correctness supersedes performance, and strong consistency thus takes precedence [26].

There is also a large class of applications which do not have a single, clear-cut goal (either performance or correctness). Instead, such applications aim to satisfy *both* of these conflicting demands. These applications fall in a *gray zone*, somewhere in-between the two previous classes, as we highlight in Figure 1. Typically, these applications aim to strike an optimal balance of consistency and performance by employing different consistency models, often at the granularity of individual operations [18, 24, 43, 51, 66]. Choosing the appropriate consistency model, even at this granularity, is hard, and

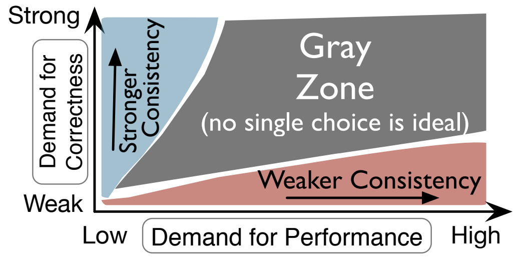


Figure 1: Many applications fall into a *gray zone*, torn between the need for both performance and correctness.

*Author names appear in alphabetical order.

the result is often sub-optimal, as developers still end up with fixing a certain side of the consistency/performance trade-off (and sacrificing the other side).

Moreover, programming in the gray area is difficult, as developers have to juggle different consistency models in their applications [24, 43]. If programming with a single consistency model (such as weak consistency [26]) is non-trivial, then mixing multiple models is even harder [50]. In their struggle to optimize performance with consistency, developers must go up against the full complexity of the underlying storage stack. This includes choosing locations (cache or backup or primary replica), dealing with coherence and cache-bypassing, or selecting quorums. These execution details reflect as a burden on developers, complicate application code, and lead to bugs [31, 55].

Our goal is to help with the programming of applications located in the gray area. We accept as a fact that no single consistency model is ideal, providing both high performance and strong consistency (correctness) at the same time [14, 33]. Our insight is to approach this ideal in complementary steps, by *combining consistency models in a single operation*. Briefly, developers can invoke an operation on a replicated object and obtain multiple, incremental *views* on the result, at successive points in time. Each view reflects the operation result under a particular consistency model. Initial (preliminary) views deliver with low latency—but weak consistency—while stronger guarantees arrive later. We call this approach *incremental consistency guarantees* (ICG).

We introduce Correctables, an abstraction which grants developers a clean, consistency-based interface for accessing replicated objects, clearly separating semantics from execution details. This abstraction reduces programmer effort by hiding storage-specific protocols, e.g., selecting quorums, locations, or managing coherence. Correctables are based on *Promises* [53], which are placeholders for a single value that becomes available in the future. Correctables generalize Promises by representing not a single, but multiple future values, corresponding to incremental views on a replicated object.

To the best of our knowledge, our abstraction is the first which enables applications to build on ICG. As few as *two* views suffice for ICG to be useful. The advantage of ICG is that applications can speculate on the preliminary view, hiding the latency of strong consistency, and thereby improving performance [71]. Speculating on preliminary responses is expedient considering that, in many systems, weak consistency provides correct results on expectation [19, 55].

Speculation with ICG is applicable to a wide range of scenarios. Consider, for instance, that a single application-level operation can aggregate multiple—up to hundreds of—storage-level objects [16, 27, 52, 65]. Since these objects are often inter-dependent, they can not always be fetched in parallel. With ICG, the application can use the fast preliminary view to speculatively prefetch any dependent objects. By the time the final (strongly consistent) view arrives, the prefetching would also finish. If the preliminary result was correct (matching the final one), then the speculation is deemed successful, reducing the overall latency of this operation.

Alternatively, ICG can open the door to exploiting application-specific semantics for optimizing performance. Imagine an application requiring a monotonically increasing counter to reach some pre-defined threshold (e.g., number of purchased items in a shop required for a fidelity discount). If a weakly consistent view of the counter already exceeds this threshold, the application can proceed without paying the latency price of a strongly consistent view.

The high-level abstraction centered on consistency models, coupled with the performance benefits of enabling speculation via ICG, are the central contributions of Correctables. We evaluate these performance benefits by modifying two well-known storage systems (Cassandra [45] and ZooKeeper [39]). We plug Correctables on top of these, build three applications (a Twissandra-based microblogging service [10], an ad serving system, and a ticket selling system), and experiment on Amazon EC2.

Our evaluation first demonstrates that there is a sizable time window between preliminary and final views, which applications can use for speculation. Second, using YCSB workloads A, B, and C, we show that we can reduce the latency of strongly consistent operations by up to 40% (from 100*ms* to 60*ms*) at little cost (10% bandwidth increase, 6% throughput drop) in the ad system. The other two applications exhibit similar improvements. Even if the preliminary result is often inconsistent (25% of accesses), incremental consistency incurs a bandwidth overhead of only 27%.

In the rest of this paper, we overview our solution in the context of related work (§2) and present the Correctables interface (§3). We show how applications use Correctables (§4), and describe the bindings to various storage stacks (§5). We then give a comprehensive evaluation (§6) and conclude (§7).

2. Overview & Related Work

This paper addresses the issue of programming and speculating with replicated objects through a novel abstraction called Correctables. In this section, we overview the main concepts behind Correctables, and we contrast our approach with related work.

2.1 Consistency Choices

There is an abundance of work on consistency models. These range from strong consistency protocols [40, 46, 68], some optimized for WAN or a specific environment [26, 29, 44, 48, 72, 74], through intermediary models such as causal consistency [30, 54], to weak consistency [28, 67]. As a recent development, storage systems offer multiple—i.e., *differentiated*—consistency guarantees [24, 43, 62]. This allows applications in the above-mentioned gray zone to balance consistency and performance on a per-operation basis: the choice of guarantees depends on how sensitive the corresponding operation is.

Differentiated guarantees can take the form of SLAs [66], policies attached to data [43], dynamic quorum selection for quorum-based storage systems such as Dynamo [28] or others [8, 45], or even ad-hoc operation invariants [18]. In practice, two consistency levels often suffice: weak and strong [1, 5]. Sensitive operations (e.g., account creation or password checking) use the strong level, while less critical operations (e.g., remove from basket) use weak guarantees [43, 66, 73] to achieve good performance.

For instance, in Gemini [51], operations are either Blue (fast, weakly consistent) or Red (slower, strongly consistent). For sensitive data such as passwords, Facebook uses a separate linearizable sub-system [55]. Likewise, Twitter employs strong consistency for "certain sets of operations" [64], and Google's Megastore exposes strong guarantees alongside read operations with "inconsistent" semantics [20]. Another frequent form of differentiated guarantees appears when applications bypass caches to ensure correctness for some operations [16, 60].

Given this great variety of differentiated guarantees, we surmise that applications can benefit from mixing consistency models. The notable downside of this approach is that application complexity increases [50]. Developers must orchestrate different storage APIs and consider the interactions between these protocols [16, 18, 69]. Our work subsumes results in this area. We propose to hide different schemes for managing consistency under a common interface, Correctables, which can abstract over a varying combination of storage tiers and reduce application complexity. In addition, we introduce the notion of *incremental consistency guarantees* (ICG), i.e., progressive refinement of the result of a *single* operation.

2.2 ICG: Incremental Consistency Guarantees

Applications which use strong consistency—either exclusively or for a few operations—do so to avoid anomalous behavior which is latent in weaker models. Interestingly, recent work reveals that this anomalous behavior is rare in practice [19, 55]. There are applications, however, which cannot afford to expose even those rare anomalies.

For instance, consider a system storing user passwords, and say it has 1% chance of exposing an inconsistent password. If such a system demands correctness—as it should—then it is forced to pay the price for strong consistency on *every* access, even though this is not necessary in 99% of cases. We propose ICG to help applications avert this dilemma, and pay for correctness only when inconsistencies actually occur.

With ICG, an application can obtain both weakly consistent (called *preliminary*) and strongly consistent (called *final*) results of an operation, one by one, as these become available. While waiting for the final result, the application can speculatively perform further processing based on the preliminary—which is correct on expectation. Following our earlier example, this would help hide the latency of strong consistency for 99% of accesses.

The full latency of strong consistency is only exposed in case of misspeculation, when the preliminary and final values diverge because the preliminary returned inconsistent data [71]. These are the 1% cases where strong consistency is needed anyway. Speculation through ICG can lessen the most prominent argument against strong consistency, namely its performance penalty. With ICG we pay the latency cost of strong consistency only when necessary, regardless of how often this is the case.

Speculation is a well-known technique for improving performance. Traditionally, the effects of speculation in a system remain hidden from higher-level applications until the speculation confirms, since the effects can lead to irrevocable actions in the applications [41, 57, 59, 71]. Alternatively, it has been shown that leaking speculative effects to higher layers can be beneficial, especially in user-facing applications, where the effects can be undone or the application can compensate in case of misspeculation [36, 47, 49, 61]. We propose to use eventual consistency as a basis for doing speculative work, as a novel approach for improving performance in replicated systems. Also, more generally, we allow the application itself (which knows best), to decide on the speculation boundary [70]—whether to externalize effects of speculation, and later to undo or compensate these effects, or whether to isolate users from speculative state.

Besides speculation, ICG is useful in other cases as well. For instance, applications can choose dynamically whether to settle with a preliminary value and forsake the final value altogether. This is a way to obtain application-specific optimizations, e.g., to enforce tight latency SLAs. Alternatively, we can *expose* the preliminary response to users and revise it later when the final response arrives. This strategy is akin to compensating in case of misspeculation, as mentioned earlier.

Clearly, not all applications are amenable to exploiting ICG. In Table 1 we give a high-level account on three categories of applications: (1) those which have no additional benefit from strong consistency or ICG; (2) those which require correct results but are not amenable to speculation; and at last (3) applications that can obtain

Category	Synopsis	Applications and use cases
Weak Consistency	Use the **weakest, but fastest** consistency model, e.g., by using partial quorums, or going to the closest replica or cache. No benefit from ICG.	Computation on static (BLOBs) content, e.g., thumbnail generator for images and videos, accessing cold data, fraud analysis, disconnected operations in mobile applications, etc.
Strong Consistency	Use the **strongest** available model, e.g., by going to the primary replica. Applications require correct results.	Infrastructure services (e.g., load-balancing, session stores, configuration and membership management services), stock tickers, trading applications, etc.
Incremental Consistency Guarantees (ICG)	Use multiple, **incremental** models. Applications benefit from weakly consistent values (e.g., by speculating or exposing them), but prefer correct results.	E-mail, calendar, social network timeline, grocery list, flight search aggregation, online shopping, news reading, browsing, backup, collaborative editing, authentication and authorization, advertising, online wallets, etc.

Table 1: Different patterns and their corresponding use cases. Many applications can benefit from ICG.

performance without sacrificing correctness by leveraging ICG.

2.3 Client-side Handling of ICG

To program with ICG, applications need to wait asynchronously for multiple replies to an operation (where each reply encapsulates a different guarantee on the result) while doing useful work, i.e., speculate. To the best of our knowledge, no abstraction fulfills these criteria. To minimize the effort of programming with ICG, we draw inspiration from *Promises*, seminal work on handling asynchronous remote procedure calls in distributed systems [53].

A Promise is a placeholder for a value that will become available asynchronously in the future. Given the urgency to handle intricate parallelism and augmenting complexity in applications, it is not surprising that Promises are becoming standard in many languages [6, 2, 12, 31]. We extend the binary interface of Promises (a value either present or absent) to obtain a multi-level abstraction, which incrementally builds up to a final, correct result.

The Observable interface from reactive programming can be seen as a similar generalization of Promises. Observables abstract over asynchronous data streams of arbitrary type and size [56]. Our goal with Correctables, in contrast, is to grant developers access to consistency guarantees on replicated objects in a simple manner. The ProgressivePromise interface in Netty [7] also generalizes Promises. While it can indicate progress of an operation, a ProgressivePromise does not expose preliminary results of this operation.

3. Correctables

This section presents the Correctables interface for programming and speculating with replicated data. Applications use this interface as a library, as Figure 2 depicts. At the top of this library sits the application-facing API. The library is connected to the storage stack using a storage binding, which is a module that encapsulates all storage

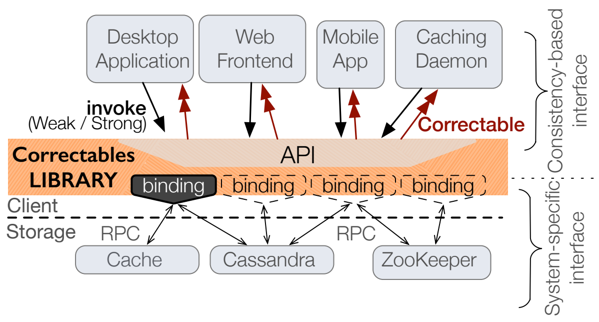


Figure 2: High-level view of Correctables, as an interface to the underlying storage.

system specific interfaces and protocols. Correctables fulfill two critical functions: (i) translate API calls into storage-specific requests via a binding, and (ii) orchestrate responses from the binding and deliver them—in an incremental way—to the application, using *Correctable* objects. Each call to an API method returns a Correctable which represents the progressively improving result (i.e., a result with ICG).

3.1 From Promises to Correctables

As mentioned earlier, Correctables descend from Promises. To model an asynchronous task, a Promise starts in the *blocked* state and transitions to *ready* when the task completes, triggering any callback associated with this state [53]. Promises help with asynchrony, but not incrementality. To convey incrementality, a Correctable starts in the *updating* state, where it remains until the final result becomes available or an error occurs (see Figure 3). When this happens, the Correctable *closes* with that result (or error), transitioning to the *final* (or *error*) state. Upon each state transition, the corresponding callback triggers. Preliminary results trigger

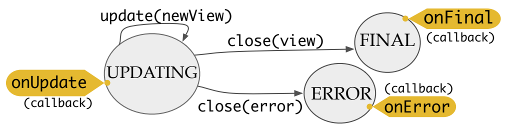


Figure 3: The three states, transitions, and callbacks associated with a Correctable.

a same-state transition (from *updating* to *updating*). A Correctable can have callbacks associated with each of its three states. To attach these callbacks, we provide the `setCallbacks` method; together with `speculate`, these two form the two central methods of a Correctable, which we examine more closely in §4.

3.2 Decoupling Semantics from Implementation

The Correctables abstraction decouples applications from storage specifics by adopting a thin, consistency-based interface, centered around *consistency levels*. This enables developers—who naturally reason in terms of consistency rather than protocol specifics—to obtain simple and portable implementations. With Correctables, applications can transparently switch storage stacks, as long as these stacks support compatible consistency models.

Our API consists of three methods:

1. `invokeWeak`(*operation*),
2. `invokeStrong`(*operation*), and
3. `invoke`(*operation*[, *levels*]).

The first two allow developers to select either weak or strong consistency for a given *operation*. The returned Correctable never transitions from *updating* to *updating* state and only closes with a final value (or error). These two methods follow the traditional practice of providing a single result which lies at one extreme of the consistency/performance trade-off.

The third method provides ICG, allowing developers to operate on this trade-off at run-time, which makes it especially relevant for applications in the above-mentioned gray area. Instead of a single result (as is the case with the two former methods), `invoke` provides incremental updates on the operation result. Optionally, `invoke` accepts as argument the set of consistency levels which the result should—one after the other—satisfy. If this argument is absent, `invoke` provides all available levels. This argument allows some optimizations, e.g., if an application only requires a subset of the available consistency levels, this parameter informs a binding to avoid using the extraneous levels; we omit further discussion of this argument due to space constraints. The available consistency levels depend on the underlying storage system and binding, which we discuss in more detail in §5.

In the next section, we show how to program with Correctables through several representative use-cases. In code snippets we adopt a Python-inspired pseudocode for readability sake. For brevity we leave aside error handling, timeouts, or other features inherited from modern Promises, such as aggregation or monadic-style chaining [12, 31, 53].

```
1  from pylons import app_globals as g # cache access
2  from r2.lib.db import queries       # backend access

4  def user_messages(user, update = False):
5    key = messages_key(user._id)
6    trees = g.permacache.get(key)
7    if not trees or update:
8        trees = user_messages_nocache(user)
9        g.permacache.set(key, trees) # cache coherence
10   return trees
11 def user_messages_nocache(user):
12   # Just like user_messages, but avoiding the cache...
```

Listing 1: Different consistency guarantees in Reddit [13], as an example of tight coupling between applications and storage. Developers must manually handle the cache and the backend.

```
def user_messages(user, strong = False):
  key = messages_key(user._id)
  # coherence handled by invoke* functions in bindings
  if strong: return invokeStrong(get(key))
      else: return invokeWeak(get(key))
```

Listing 2: Reddit code rewritten using Correctables.

4. Correctables in Action

This section presents examples of how Correctables can be useful on two main fronts. (1) Decoupling applications from their storage stacks by providing an abstraction based on consistency levels. (2) Improving application performance by means of ICG, e.g., via speculation or exploiting application-specific semantics.

4.1 Decoupling Applications from Storage

We first discuss a simple case of decoupling, where we illustrate the use the first two functions in our API, namely `invokeWeak` and `invokeStrong`. As discussed in §2, many applications differentiate between weak and strong consistency to balance correctness with performance. In practice, applications often resort to ad-hoc techniques such as cache-bypassing to achieve this, which complicates code and leads to errors [16, 31]. Listing 1 shows code from Reddit [13], a popular bulletin-board system and a prime example of such code. Developers have to explicitly handle cache access (lines L6 and L9), make choices based on presence of items in the cache (L7), manually bypass the cache (L8) under specific conditions, and write duplicate code (L12).

Instead of explicit cache-bypassing, we can employ `invokeWeak` and `invokeStrong` to substantially simplify the code by replacing ad-hoc abstractions like `user_messages` and `user_messages_nocache`, as Listing 2 shows. Furthermore, we can replace other near-identical functions for differentiated guarantees, elimi-

```
1 invoke(read(...))
2   .speculate(speculationFunc[, abortFunc])
3   .setCallbacks(onFinal = (res) => deliver(res))
```

Listing 3: Generic speculation with Correctables. The square brackets indicate that `abortFunc` is optional.

nating duplicate logic.[1] Cache-coherence and bypassing is completely handled by the storage-specific binding. This reduces both programmer effort and application-level complexity.

The third method in our library is `invoke`. Correctables are crucial for this method, since it captures ICG. `invoke` allows applications to speculate on preliminary values (hiding the latency of strong consistency), or exploit application-specific semantics, as we show next.

4.2 Speculating with Correctables

Many applications are amenable to speculating on preliminary values to reap performance benefits. To understand how to achieve this, we consider any non-trivial operation in a distributed application which involves reading data from storage. Using `invoke` to access the storage, applications can perform speculation on the preliminary value. If this preliminary value is confirmed by the final value, then speculation was correct, reducing overall latency [71]. Examples where speculation applies include password checking or thumbnail generation (as mentioned in [66]), as well as operations for airline seat reservation [73], or web shopping [43].

Listing 3 depicts how this is performed in practice with Correctables. Even though such speculation can be orchestrated directly by using the `onUpdate` and `onFinal` callbacks of a Correctable object, we provide a convenience method called `speculate` that captures the speculation pattern (L2). It takes a speculation function as an argument, applying it to every new view delivered by the underlying Correctable if this view differs from the previous one. The `speculate` method returns a new Correctable object which closes with the return value of the user-provided speculation function. If the final view matches a preliminary one (which is the common case), the new Correctable can close immediately when the final view becomes available, confirming the speculation. Otherwise, it closes only after the speculation function is (automatically) re-executed with correct input. In the latter case, an optional abort function is executed, undoing potential side-effects of the preceding speculation. Next, we discuss an ad serving system as an example application that can benefit from such speculation.

[1]Similar pairs of ad-hoc functions exist in Reddit for accessing other objects. Perhaps accidentally, these other functions contain comments referring to `user_messages` instead of their specific objects. We interpret this as a strong indication of "copy-pasting" code, which Correctables would help prevent.

```
1 def fetchAdsByUserId(uid):
2   invoke(getPersonalizedAdsRefs(uid))
3     .speculate(getAds) # fetch & post−process ads
4     .setCallbacks(onFinal = (ads) => deliver(ads))
```

Listing 4: Example of applying speculation in an advertising system to hide latency of strong consistency.

Advertising System. Typically, ads are personalized to user interests. These interests fluctuate frequently, and so ads change accordingly [42]. Given their revenue-based nature, advertising systems have conflicting requirements, as they aim to reconcile consistency (freshness of ads) with performance (latency) [24, 26]. We thus find that they correspond to our notion of gray area, and are a suitable speculation use-case.

Listing 4 shows how we can use ICG while fetching ads. First, we obtain a list of *references* to personalized ads using the `invoke` method (L2). This method returns both a preliminary view (with weak guarantees) and a final (fresh) view. Using the references in the preliminary view, we fetch the actual ads content and media, and do any post-processing, such as localization or personalization (L3). If the final view corresponds to the preliminary, then speculation was correct, and we can deliver (L4) the ads fast; otherwise, `getAds` re-executes on the final view, and we deliver the result later. We use this application as our first experimental case-study (§6.3.2).

The pattern of fetching objects based on their references—which themselves need to be fetched first—is widespread. It appears in many applications, such as reading the latest news, the most recent transactions, the latest updates in a social network, an inventory, the most pressing items in a to-do list or calendar, and so on. In all these cases, the application needs to chase a pointer (reference) to the latest data, while weak consistency can reveal stale values, which is undesirable. We avoid stale data by reading the references with `invoke`, and we mask the latency of the final value by speculatively fetching objects based on the preliminary reference.

4.3 Exploiting Application Semantics

Applications can exploit their specific semantics to leverage the preliminary and the final values of `invoke`. For instance, consider the web auction system mentioned by Kraska et al. [43], where strong consistency is critical in the last moments of a bid, but is not particularly helpful in the days before the bid ends, when contention is very low and anomalous behavior is unlikely. Another example is selling items from a predefined stock of such items. If a preliminary response suggests that the stock is still big, it is safe to proceed with a purchase. Otherwise, if the stock is almost empty, it would be better to wait for the arrival of the final response. This is the case, for instance, for a system selling tickets to an event, which we describe next.

```
def purchaseTicket(eventID):
  done = false
  invoke(dequeue(eventID)).setCallbacks(
    onUpdate = (weakResult) =>
      if weakResult.ticketNr > THRESHOLD:
        done = true # many tickets left , so we can buy
        confirmPurchase()
    onFinal = (strongResult) =>
      if not done and strongResult is not null:
        confirmPurchase() # we managed to get a ticket
      else: display("Sold out. Sorry!"))
```

Listing 5: Dynamic selection of consistency guarantees in a ticket selling system. If there are many tickets in the stock, we can safely use weak consistency.

Selling Tickets for Events. For this application system, we depart from the popular key-value data type. First, as we want to avoid overselling, we need a stronger abstraction to serialize access to the ticket stock. Simple read/write objects (without transactional support) are fundamentally insufficient [37]. Second, we want to demonstrate the applicability of ICG to other data types. We thus model the ticket stock using a queue, which is a simple object, yet powerful enough to avoid overselling.

Event organizers enqueue tickets and retailers dequeue them. This data type allows us to serialize access to the shared ticket stock [15, 43]. We assume, however, that tickets bear no specific ordering (i.e., there is no seating). Clients are interested in purchasing *some* ticket, and it is irrelevant which exact element of the queue is dequeued. We can thus resort to weak consistency most of the time, and use strong consistency sparingly. We consider a weakly consistent result of an operation to be the outcome of simulating that operation on the local state of a single replica (see §5.2).

Listing 5 shows how we can selectively use strong consistency in this case, based on the estimated stock size. For each purchase, retailers use `invoke` with the dequeue operation. This yields a quick preliminary response, by peeking at the queue tail on the closest replica of the queue. If the preliminary value indicates that there are many tickets left (e.g., via a ticket sequence number, denoting the ticket's position in the queue), which is the common case, the purchase can succeed without synchronous coordination on dequeue, which completes in the background. This reduces the latency of most purchase operations. As the queue drains, e.g. below a predefined threshold of 20 tickets, retailers start waiting for the final results, which gives atomic semantics on dequeuing, but incurs higher latency. This system represents our second experimental case study (§6.3.2).

4.4 Exposing Data Incrementally

In some cases, it is beneficial to expose even incorrect (stale) data to the user if this data arrives fast, and amend the output as more fresh data becomes available. Indeed, a quick approximate result is sometimes better than

```
invoke(getLatestNews()).setCallbacks(
    onUpdate = (items) => refreshDisplay(items))
```

Listing 6: Progressive display of news items using Correctables. The `refreshDisplay` function triggers with every update on the news items.

an overdue reply [28, 66]. Many applications update their output as better results become available. A notable example is flight search aggregators [9], or generally, applications which exhibit high responsiveness by leaking to the user intermediary views on an ongoing operation [47, 49], e.g., previews to a video or shipment tracking. We can assist the development of this type of applications, as we describe next.

Smartphone News Reader. Consider a smartphone news reader application for a news service replicated with a primary-backup scheme [66]. Additionally, recently seen news items are stored in a local phone cache. With ICG provided by Correctables, the application can be oblivious to storage details. It can use a single logical storage access to fetch the latest news items, as Listing 6 shows. The binding would translate this logical access to three actual requests: one to the local cache, resolving almost immediately, one to the closest backup replica, providing a fresher view, and one to a more distant primary replica, taking the longest to return but providing the most up-to-date news stories.

4.5 Discussion: Applicability of ICG

In a majority of use-cases, we observe that two views suffice. Correctables, however, support arbitrarily many views. Note that this does not add any complexity to the interface and can be useful, as the news reader application shows.

There are other examples of applications which can benefit from multiple views. A notable use-case are blockchain-based applications (e.g., Bitcoin [58]), where Correctables can track transaction confirmations as they accumulate and eventually the transaction becomes an irrevocable part of the blockchain, i.e., strongly-consistent with high probability. This is a use-case we also implemented, but omit for space constraints. In larger quorum systems (e.g., BFT), Correctables can represent the majority vote as it settles. Search or recommenders, likewise, can benefit from exposing multiple intermediary results in subsequent updates.[2]

Intuitively, multiple preliminary views are helpful for applications requiring live updates. On the one hand, several preliminary values would make the application more interactive and offer users a finer sense of progress. This is especially important when the final result has high latency (Bitcoin transactions take tens of minutes). On the other hand, as the replicated system delivers more

[2]We are grateful to our anonymous OSDI reviewers for this particularly constructive idea.

preliminary views for an operation, less operations can be sustained and overall throughput drops. Thus, applications which build on ICG with multiple incremental views observe a trade-off between interactivity and throughput. This trade-off can be observed even when the system delivers only two views (§6.2.1).

In order to be practical, the cost of generating and exploiting the preliminary values of ICG must not outweight their benefits. The cost of generating ICG is captured in the trade-off we highlighted above; the cost of exploiting ICG is highly application-dependent. If used for speculation, the utility of 2+ views depends on how expensive it is to re-do the speculative work upon misspeculation. This can range from negligible (simply display preliminary views) to potentially very expensive (prefetch bulky data). Additionally, the utility also depends on how often misspeculation actually occurs. This depends on the workload characteristics: workloads with higher write ratios elicit higher rates of inconsistencies, and thus more misspeculations (§6.2.1–Divergence).

There are also cases when using ICG is not an option. This is either due to the underlying storage providing a unique consistency model and lacking caches, or due to application semantics, which can render ICG unnecessary—we give examples of this in the first two rows of Table 1. Correctables, however, are beneficial beyond ICG. This abstraction can hide the complexity of dealing with storage-specific protocols, e.g., quorum-size selection. The application code thus becomes portable across different storage systems.

5. Bindings

Our library handles all the instrumentation around Correctable objects. This includes creation, state transitions, callbacks, and the API inherited from Promises [12, 31]. Bindings are storage-specific modules which the library uses to communicate with the storage. These modules encapsulate everything that is storage system specific, and thus draw the separating line between consistency models—which Correctables expose—and implementations of these models. In this section, we describe the binding API, and show how bindings can facilitate efficient implementation of ICG with server-side support.

5.1 Binding API

An instance of our library always uses one specific binding. A binding establishes: (1) the concrete configuration of the underlying storage stack (e.g., Memcache on top of Cassandra) together with (2) the *consistency levels* offered by this stack, and (3) the implementation of any storage specific protocol (e.g., for coherence, choosing quorums). This allows the library to act as a client to the storage stack.

When an application calls an API method (§3.2), the library immediately returns a Correctable. In the background, we use the *binding API* to access the underlying storage. The binding forwards responses from the storage through an upcall to the library. The library then updates (or closes) the associated Correctable, executing the corresponding callback function.

The binding API exposes two methods to the library. First, `consistencyLevels()` advertises to the library the supported consistency levels. It simply returns a list of supported consistency levels, ordered from weakest to strongest. In most implementations, this will probably be a one-liner returning a statically defined list. The second function is `submitOperation(op, consLevels, callback)`. The library uses this function to execute operation `op` on the underlying storage, with `consLevels` specifying the requested consistency levels. The `callback` activates whenever a new view of the result is available. The binding has to implement the protocol for executing `op` and invoke `callback` once for each requested consistency level.

Listing 7 shows the implementation of a simple binding for a primary-backup storage, supporting two consistency levels. A more sophisticated binding could access the backup and primary in parallel, or could provide more than two consistency levels. We designed the binding API to be as simple as possible; contributors or developers wishing to support a particular store must implement this API when adding new bindings. We currently provide bindings to Cassandra and ZooKeeper.

5.2 Efficiency and Server-side Support

On a first glance, ICG might seem to evoke large bandwidth and computation overheads. Indeed, if the `invoke` method comprises multiple independent single-consistency requests, then storage servers will partly redo their own work. Also, as the weakly and strongly consistent values often coincide, multiple responses are frequently redundant. Such overheads would reduce the practicality of ICG.

```
def consistencyLevels():
  return [WEAK, STRONG]

def submitOperation(operation, consLevels, callback):
  if WEAK in consLevels:
    backupResult = queryClosestBackup(operation)
    callback(backupResult, WEAK)
  if STRONG in consLevels:
    primaryResult = queryPrimary(operation)
    callback(primaryResult, STRONG)
```

Listing 7: Simple binding to a storage system with primary-backup replication.

With server-side support, however, we can minimize these overheads. For instance, we can send a *single* request to obtain all the incremental views on a replicated object. An effective way to do this is to hook into the coordination mechanism of consistency protocols. This mechanism is the core of such protocols, and the provided consistency model and latency depend on the type of coordination. For example, asynchronous (off the critical path) coordination ensures eventually consistent results with low-latency [28]. Coordination through an agreement protocol, as in Paxos [46], yields linearizability [38], but with a higher latency.

Our basic insight is that we can get a good guess of the result already before coordinating, based on a replica's local state. In fact, this same state is being exposed when asynchronous coordination is employed, and as we alreay mentioned, this state is consistent on expectation. The replica can leak a preliminary response—with weak guarantees—to the client prior to coordination (Figure 4). Moreover, we can reduce bandwidth overhead by skipping the final response if it is the same as the preliminary: a small *confirmation* message suffices, to indicate that the preliminary response was correct. Indeed, with such an optimization, ICG has minor bandwidth overhead (§6.2.1).

An additional benefit from this approach compared to sending two independent requests is that it prevents certain types of unexpected outcomes. For instance, strong consistency might be more stale than weak consistency if responses to two independent requests were reordered by the WAN [66]. Using this approach, we modify two popular systems—Cassandra and ZooKeeper—to provide efficient support for ICG. Other techniques (e.g., master leases [23]) or replication schemes (e.g., primary-backup) can provide final views fast, skipping the preliminary altogether.

Cassandra. Cassandra uses a quorum-gathering protocol for coordination [32]. In our modified version of Cassandra—called Correctable Cassandra (CC)—the coordinating node sends a preliminary view after obtaining the *first* result from any replica. This view has low latency, obtained either locally (if the coordinator is itself a replica) or from the closest replica. Our binding

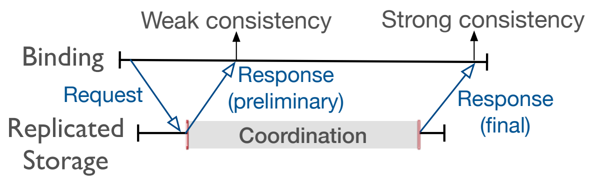


Figure 4: Simple server support for efficient ICG. The storage system sends a preliminary response before coordinating. Note that for a single request, the storage provides two responses.

to CC supports two consistency levels, *weak* (involving one replica) and *strong* (involving two or more). To minimize bandwidth overhead of `invoke`, CC uses the confirmation messages optimization we mentioned earlier.

ZooKeeper. To demonstrate the versatility of Correctables, we consider a different data type, namely replicated queues, which ZooKeeper can easily model [11]. Our binding supports operations `enqueue` and `dequeue`, with weak and strong consistency semantics, accessible via `invokeWeak` and `invokeStrong`, respectively; `invoke` supplies both consistency models incrementally.

The vanilla ZooKeeper implementation (ZK) has strong consistency [39]. For efficient ICG, we implement Correctable ZooKeeper (CZK) by adding a fast path to ZK: a replica first simulates the operation on its local state, returning the preliminary (weak) result. After coordination (via the Zab protocol [40]), this replica applies the operation and returns the strong response.

Causal Consistency and Caching. We also implement a binding to abstract over a causally consistent store complemented by a client-side cache. The `invoke` function reveals two views: one from cache (very fast, possibly stale), and another from the causally consistent store. This binding ensures write-through cache coherence, allows cache-bypassing (`invokeStrong`) or direct cache access (`invokeWeak`), e.g., in case of disconnected operations for mobile applications [62]. Given the space constraints we focus on the two other bindings.

6. Evaluation

Our evaluation focuses on quantifying the benefits of ICG. Before diving into it, it is important to note that any potential benefit of ICG is capped by performance gaps among consistency models. Briefly, if strong consistency has the same performance as weaker models (or the difference is negligible) then applications can directly use the stronger model. This is, however, rarely the case. In practice, there can be sizable differences—up to orders of magnitude—across models [17, 66].

We first describe our evaluation methodology, and then show that such optimization potential indeed exists. We do so by looking at the performance gaps between weak and strong consistency in quorum-based (Cassandra) and consensus-based (ZooKeeper) systems. We then quantify the performance gain of using ICG in three case studies: a Twissandra-based microblogging service [10], an ad serving system, and a ticket selling application.

6.1 Methodology

We run all experiments on Amazon's EC2 with m4.large instances and a replication factor of 3, with replicas distributed in Frankfurt (FRK), Ireland (IRL), and N. Virginia (VRG). Unless stated otherwise, to obtain WAN conditions, the client is in IRL and uses the replica in

FRK; note that colocating the client with its contact server (i.e., both in IRL) would play to our advantage, as it would reduce the latency of preliminary responses and allow a bigger performance gap. We also experiment with various other client locations in some experiments.

For Cassandra experiments, we compare the baseline Cassandra v2.1.10 (labeled C), with our modified Correctable Cassandra (CC). We use superscript notation to indicate the specific quorum size for an execution, e.g., C^1 denotes a client reading from Cassandra with a read quorum $R = 1$ (i.e., involving 1 out of 3 replicas). For the ZooKeeper queue, we compare our modified Correctable ZooKeeper (CZK) against vanilla ZooKeeper (ZK), v3.4.8. The cumulative implementation effort associated with CC and CZK, including three case studies, is modest, at roughly $3k$ lines of Java code.

6.2 Potential for Exploiting ICG

To determine the potential of ICG, we examine their behavior in practice. Studies show that large load on a system and high inter-replica latencies give rise to large performance gaps among consistency models [17, 66]. To the best of our knowledge, however, there are no studies which consider a combination of incremental consistency models in a single operation. We first investigate this behavior in Cassandra and then in ZooKeeper.

6.2.1 Potential for Exploiting ICG in Cassandra

Cassandra can offer us insights into the basic behavior of ICG in a quorum system. As explained in §5, CC offers two consistency models: weak, which yields the *preliminary* view ($R = 1$), and strong, giving the *final* view ($R = 2$ or $R = 3$, depending on the requested quorum size). For write operations, we set $W = 1$. We use microbenchmarks and YCSB [25] to measure single-request latency and performance under load, respectively. For each CC experiment, we run three 60-second trials and elide from the results the first and last 15 seconds. We report on the average and 99th percentile latency, omitting error bars if negligible.

Single-request Latency. We use a microbenchmark consisting of read-only operations on objects of $100B$. We are interested in the performance gap between preliminary and final views as provided by ICG, and we contrast these with their vanilla counterparts. We thus compare CC^2 ($R \in \{1,2\}$) and CC^3 ($R \in \{1,3\}$) with C^1 ($R = 1$), C^2 ($R = 2$), and C^3 ($R = 3$). For CC, R has two values: the read quorum size for the preliminary (weak) and for the final (strong) replies, respectively.

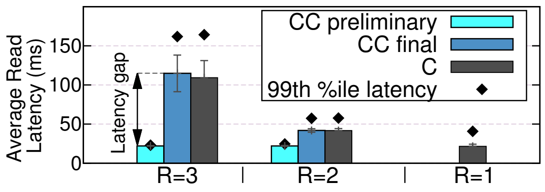


Figure 5: Single-request latencies in Cassandra for different quorum configurations. A bigger latency gap means a larger time window available for speculation.

Figure 5 shows the results for all these configurations, grouped by their read quorum size. The average latency of preliminary views—whether it is for CC^2 or CC^3—follows closely the latency of C^1, which coincides with the $20ms$ RTT between the client and the coordinator. Preliminary views reflect the local state on the replica in FRK, having the same consistency as C^1. Final views of CC^2 and CC^3 follow the trend of the requested quorum size and reflect the behavior of C^2 and C^3 respectively.

The performance gap between the preliminary and final view for CC^2 is $20ms$. The coordinator (FRK) is gathering a quorum of two: itself and the closest replica (IRL). The gap indeed corresponds to the RTT between these two regions. For CC^3, the gap is much larger: up to $140ms$ for the 99th percentile, due to the larger distance to reach the third replica (VRG). By speculating on the preliminary views, applications can hide up to $20ms$ (or $140ms$) of the latency for stronger consistency. In practice, such differences already impact revenue, as users are highly-sensitive to latency fluctuations [28, 35].

Performance Under Load. We also study the performance gap using YCSB workloads A (50:50 read/write ratio), B (95:5 read/write ratio), and C (read-only) [25]. To stress the systems and obtain WAN conditions, we deploy 3 clients, one per region, with each client connecting to a remote replica. For brevity, we only report on the results for the client in IRL and $R = \{1,2\}$. Figure 6 presents the average latency as a function of throughput. We plot the evolution of both the preliminary and final views individually.

We observe that CC trades in some throughput due to the load generated on the coordinator, which handles ICG. We observe this behavior in all three workloads. This is to be expected, considering the modifications necessary to implement preliminary replies (§5.2). Briefly, we add another step to every read operation that uses quorums larger than one. This step, called *preliminary flushing*, occurs at any coordinator replica serving read operations as soon as that replica finishes reading the requested data from its local storage—and prior to gathering a quorum from other replicas. This step generates additional load on the coordinator replica, explaining the throughput drop of CC^2 compared to baselines. Related work on replicated state machines (RSM) suggests an optimization [71] which resembles our flushing technique. Perhaps unsurprisingly, the optimized RSM exhibits a similar throughput drop [71, §6.2] as we notice in these experiments.

The latency gap between preliminary and final views

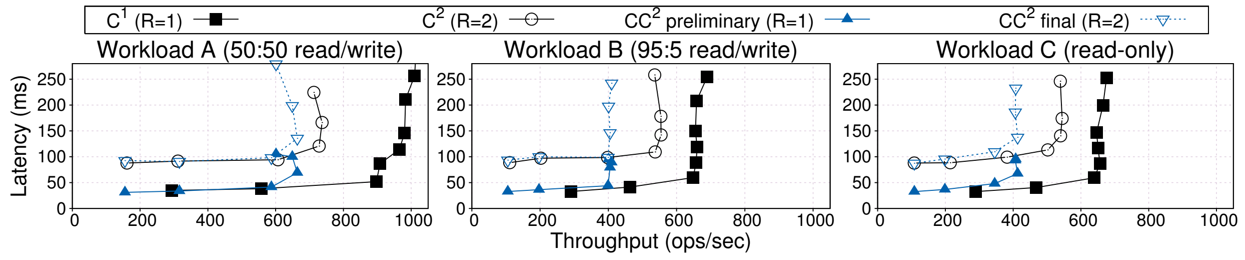


Figure 6: Performance of Correctable Cassandra (CC) compared to baseline Cassandra (C). Note that the measurements for CC^2 have two results, one for the preliminary view and another for final. These two have the same throughput but different latencies.

is the same as the one we observe in the microbenchmarks. To conclude, our results confirm that the performance gaps while using ICG are noticeable, and hence there is room for hiding latency.

Divergence. To obtain more insight about the behavior of ICG, we use CC and the YCSB benchmark to measure how often preliminary values diverge from final results. We achieve this by using `invoke` and comparing the preliminary view to the final one. We run this experiment with a small dataset of $1K$ objects. We aim at obtaining the conditions of a highly-loaded system where clients are mostly interested in a small (popular) part of the dataset.

Figure 7 shows our result for a mix of representative YCSB workloads (A and B) and access patterns (Zipfian and Latest) with default settings. Notably, workload A (50:50 read/write) under Latest distribution (read activity skewed towards recently updated items) exhibits high divergence, up to 25%. Under such conditions, using $R = 1$ would yield many stale results. Indeed, some applications with high write ratios, e.g., notification or session stores [25, 34], tend to use $R = 2$, even though this forces *all* read operations to pay the latency price [19].

In fact, even if less than 1% of accessed objects are inconsistent, these are typically the most popular ("linchpin" [16, 60]) objects, being both read- and write-intensive. Such anomalies have a disproportionate effect at application-level, since they reflect in many more than 1% application-level operations. Applications with high update ratios as modeled by workload A, e.g., social networks [24], can thus benefit from exploiting ICG to avoid anomalies.

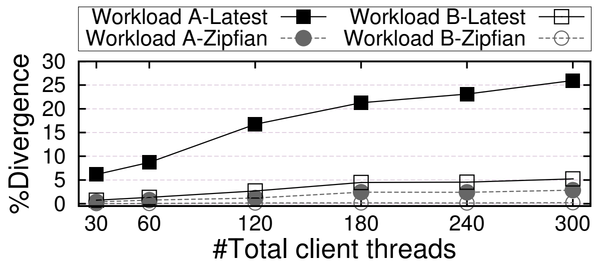


Figure 7: Divergence of preliminary from final (correct) views in Correctable Cassandra with various YCSB configurations.

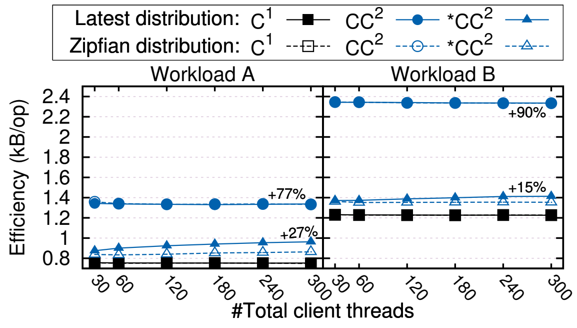


Figure 8: Efficiency (bandwidth overhead) of the ICG implementation in Correctable Cassandra (CC).

Bandwidth Overhead. In addition to the throughput drop mentioned above, client-replica bandwidth is the next relevant metric which ICG can impact. Yet, optimizations can cut the cost of this feature (§5.2). We implement such an optimization in CC, whereby a final view contains only a small confirmation—instead of the full response—if it coincides with the preliminary view. We note that in all experiments thus far we did not rely on this optimization, which makes our comparisons with Cassandra conservative.

To obtain a worst-case characterization of the costs of ICG, we consider the scenario where divergence can be maximal, as this will lessen the amount of bandwidth we can save with our optimization. Hence, we consider the exact conditions we use in the divergence benchmark, where we discovered that divergence can rise up to 25%. In this experiment, we measure the average data transferred (KB) per operation. We contrast three scenarios. First, as baseline, we use C^1, where clients request a single consistency version using weak reads. The other two systems are CC^2 (without optimization) and $^*CC^2$ (optimized to reduce bandwidth overhead).

Figure 8 shows our results. As expected, if divergence is very high—notably in workload A—then many preliminary results are incorrect. This means that final views cannot be replaced by confirmations, increasing the data cost by up to 27%. Without any optimization, this would drive the cost up by 77%. Workload B has a smaller write ratio (5%), so a lower divergence and more optimization potential: we can reduce the overhead from 90% down

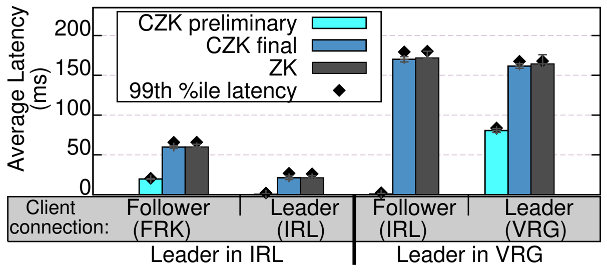


Figure 9: Latency gaps between preliminary and final views on the result of dequeue operations in Correctable ZooKeeper (CZK) compared to ZooKeeper (ZK). Client is in IRL.

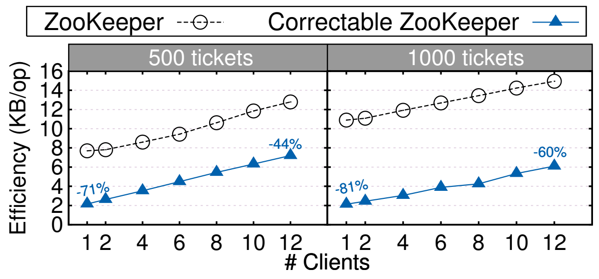


Figure 10: Efficiency (bandwidth overhead) for dequeuing operation in Correctable ZooKeeper (CZK) and ZooKeeper (ZK). Overhead in CZK is independent of queue size.

to 15% (since most final views are confirmations).

Our experiments prove that ICG have a modest cost in terms of data usage. This cost can be further reduced through additional techniques (§5.2). We remark that our choice of baseline, C^1, is conservative, because CC^2 offers better guarantees than C^1. A different baseline would be a system where clients *send two requests*—one for $R = 1$ and one for $R = 2$—and *receive two replies*. While such a baseline offers the same properties as CC^2, it would involve bigger data consumption, putting our system at an advantage.

6.2.2 Potential for Exploiting ICG in ZooKeeper

Latency Gaps. We also measure performance gaps in ZooKeeper queues for various locations of the leader and the replica which the client (in IRL) connects to. We show the results for four representative configurations for adding elements to a queue (we discuss dequeuing in the context of a ticket selling system in §6.3.2). The elements are small, containing an identifier of up to 20B (e.g., ticket number). Figure 9 shows the latency gaps when we use ICG in Correctable ZooKeeper (CZK) compared to baseline ZooKeeper (ZK).

In all cases, the latency of the preliminary view (containing the name of the assigned znode) corresponds to the RTT between the client and the contacted replica. This latency ranges from 2ms (when client and replica are both in IRL), through 20ms (the RTT from IRL to FRK), up to 83ms (the RTT between IRL and VRG). The most appealing part of this result is perhaps the substantial gap which appears when the client and the closest follower are in IRL and the leader is distant (in VRG), in the third group of results in Figure 9.

Bandwidth Overhead. Storing big chunks of data is not ZooKeeper's main goal. The client-server bandwidth is usually not dominated by the payload, reducing the benefits of the confirmation optimization. For enqueuing, the bandwidth cost thus increases by roughly 50%, from 270 to 400 bytes/operation. As expected, this corresponds to one additional (preliminary) response message in addition to the original request and (final) response.

While queues are a common ZooKeeper use-case, a problem appears in standard dequeue implementations due to message size inflation [3]. Specifically, clients first read the *whole queue* and then try to remove the tail element. To evade this problem in CZK, clients only read the constant-sized tail relevant for dequeuing. Figure 10 compares the bandwidth cost per dequeue operation in CZK and ZK for different queue sizes as we increase the number of contending threads. While the cost still increases with contention in both cases, in CZK we make it independent of queue size, which is not the case for ZK. As future work, we plan to make the dequeue cost also independent of contention using tombstones [63].

6.3 Case Studies for Exploiting ICG

Given the optimization potential explored so far, we now investigate how to exploit it in the context of three applications: the Twissandra microblogging service [10], an ad serving system, and a ticket selling system. The first two build on CC and use speculation. The last application uses CZK queues.

6.3.1 Speculation Case Studies

For Twissandra, we are interested in `get_timeline` operation, since this is a central operation and is amenable to optimization through speculation. This operation proceeds in two-steps: (1) fetch the timeline (tweet IDs), and then (2) fetch each tweet by its ID. We re-implement this function to use `invoke` on step (1) and leverage the preliminary timeline view to speculatively execute step (2) by prefetching the tweets. If the final timeline corresponds to the preliminary, then the prefetch was successful and we can reduce the total latency of the operation. In case the final timeline view is different, we fetch the tweets again based on their IDs from this final view.

Our second speculation case study is the ad serving system we describe in §4.2. The goal is to reduce the total latency of `fetchAdsByUserId` operation without sacrificing consistency, so we exploit ICG by speculating on preliminary values (Listing 4).

For both systems, we adapt their respective operations to use `invoke` ($R = \{1,2\}$) and plug them in the YCSB framework. We compare these operations using a baseline that uses only the strongly consistent result ($R = 2$),

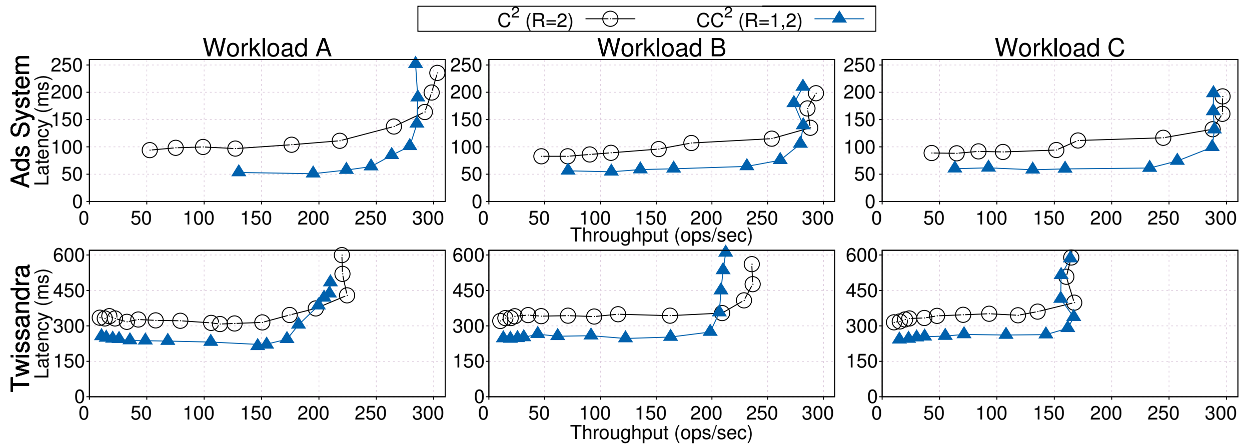


Figure 11: Using speculation via ICG to improve latency in the advertising system and in Twissandra (`get_timeline` operation). Correctable Cassandra (CC) improves latency by up to 40% in exchange for a throughput drop of 6%.

and does not leverage speculation. For Twissandra we use a corpus of 65k tweets [4] spread over 22k user timelines; the ad serving system uses a dataset of 100k user-profiles and 230k ads, where each profile references between 1 and 40 random ads.

The results are in Figure 11. In contrast to our other experiments, we deploy Twissandra replicas in Virginia, N. California, and Oregon EC2 regions. The goal is to see how performance gains vary based on deployment scenario. The ads system uses the same configuration as before. The client is in IRL for both experiments.

We first explain the results for the ads system. As can be seen, these are consistent with our earlier findings from Cassandra experiments (Figure 6). We trade throughput for better latency. Prior to saturation, we can serve ads with an average latency of 60ms. In the same conditions, the baseline achieves 100ms average latency (improvement by 40%). In turn, the throughput drop is most noticeable in workload A, by 18ops/sec (reduced by 6%). The smaller throughput drop compared to the raw results of Figure 6 is explained by the fact that each `fetchAdsByUserId` entails two storage accesses. Only the first access, however, uses ICG (to speculate). The second storage access is hidden inside `getAds` (Listing 4, L3); this is a read with $R = 2$, incurring no extra cost.

For Twissandra, we observe a lower throughput and higher latency, as the client is farther from the coordinator and replicas are also more distant from each other. But otherwise we draw similar conclusions. Notably, across both of these case-studies, divergence was consistently under 1%, so the applications encountered very few misspeculations.

6.3.2 Selling Tickets to Events

A second notable use-case of ICG is exploiting application semantics, as we discuss in the ticket selling system from §4.3 (see Listing 5). Here we exploit the fact that

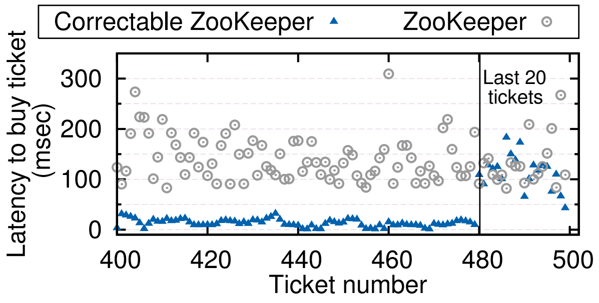


Figure 12: Selling tickets with ZK and CZK. The last 20 tickets incur high latency due to strong consistency.

the position of a ticket in the queue is irrelevant. Thus, in the common case, we can rely on the preliminary value. Strong consistency (atomicity), however, becomes critical when ticket retailers are contending over the last few remaining tickets. Using ICG, we can switch dynamically between using the preliminary or the final results when the stock becomes low, to avoid overselling.

We consider 4 retailers concurrently serving (dequeuing) tickets from a fixed-size stock of 500 tickets. Retailers are colocated with a CZK follower in FRK, the leader being in IRL. We wait for the final (atomic, equivalent to ZK) response for the last 20 tickets, otherwise we use the preliminary one. This is a conservative bound; in our experiments, only the last two tickets were “revoked” by the final view on average, with a maximum of six.

Figure 12 shows individual ticket purchase latencies, averaged over five runs, compared to latencies with vanilla CZK. As long as there are more than 20 tickets left, we reduce the purchase latency substantially. The high variability of final view latencies is caused by contention between the retailers, which does not affect preliminary views. We experiment also with larger ticket stocks (1000), but the queue length has no practical effect on latencies. To support more contention (more retailers) in practice, such a ticketing service can scale-out. For instance, we can shard the ticket stock and instantiate multiple replicated CZK services, each of them serving a partition of the overall stock, ensuring scalability [22].

7. Conclusions

We have presented Correctables, an abstraction for programming with replicated objects. The contribution of Correctables is twofold. First, they *decouple* an application from its underlying storage stack by drawing a clear boundary between consistency guarantees and the various methods of achieving them. This reduces developer effort and allows for simpler and more portable code.

Second, Correctables provide *incremental consistency guarantees* (ICG), which allow to compose multiple consistency levels within a single operation. With this type of guarantees we aim to fill a gap in the consistency/performance trade-off. Namely, applications can make last-minute decisions about what consistency level to use in an operation while this operation is executing. This opens the door to new optimizations based on speculation or on concrete, application-specific semantics.

We evaluated the performance and overhead of ICG, as well as the impact of this novel type of guarantees on three practical systems: (1) a microblogging service and (2) an ad serving system, both backed by Cassandra, and (3) a ticket selling system based on ZooKeeper queues. We modified both Cassandra and ZooKeeper to support ICG with little overhead. We showed how ICG provided by Correctables bring substantial latency decrease for the price of small bandwidth overhead and throughput drop.

We believe that Correctables provide a new way to structure the interaction between applications and their storage by exploiting incrementality, and hence a new way to build distributed applications.

Acknowledgements

We thank our shepherd, Timothy Roscoe, and the anonymous OSDI reviewers for their thoughtful comments which greatly improved the quality of our paper. We are also grateful to our colleagues from the Distributed Programming Laboratory (LPD) for putting up with our recurring requests for feedback, and for the insightful discussions we had along the way with Martin Odersky (who also suggested us the name *Correctables*), Edouard Bugnion, Willy Zwaenepoel, John Wilkes, Aleksandar Dragojević, Julia Proskurnia, Vlad Ureche, and Jad Hamza. A special thanks goes to Kenji Relut for his help with ZooKeeper. This work has been supported in part by the European ERC Grant 339539 - AOC and the Swiss FNS grant 20021_147067.

References

[1] Amazon SimpleDB. https://aws.amazon.com/simpledb/.

[2] C++ Futures at Instagram. http://instagram-engineering.tumblr.com/post/121930298932/c-futures-at-instagram.

[3] Distributed queue. netflix/curator. https://github.com/Netflix/curator/wiki/Distributed-Queue.

[4] Followthehashtag / 170,000 Apple tweets. http://followthehashtag.com/datasets/170000-apple-tweets-free-twitter-dataset/.

[5] Google appengine. https://appengine.google.com/.

[6] google/guava wiki: ListenableFutureExplained. https://github.com/google/guava/wiki/ListenableFutureExplained.

[7] ProgressivePromise (Netty 4.0 API). http://netty.io/4.0/api/io/netty/util/concurrent/ProgressivePromise.html.

[8] Riak KV, distributed NoSQL database. http://basho.com/products/riak-kv/.

[9] Skyscanner. http://www.skyscanner.ch.

[10] Twissandra. https://github.com/twissandra/twissandra/.

[11] ZooKeeper Recipes and Solutions. http://tiny.cc/zkqueues.

[12] Futures for C++11 at Facebook, 2015. https://code.facebook.com/posts/1661982097368498.

[13] reddit/r2/r2/lib/comment_tree.py:308, Accessed March, 2016. Source: https://github.com/reddit/reddit.

[14] D. J. Abadi. Consistency tradeoffs in modern distributed database system design: CAP is only part of the story. *Computer*, (2), 2012.

[15] M. K. Aguilera, A. Merchant, M. Shah, A. Veitch, and C. Karamanolis. Sinfonia: A new paradigm for building scalable distributed systems. In *SOSP*, 2007.

[16] P. Ajoux, N. Bronson, S. Kumar, W. Lloyd, and K. Veeraraghavan. Challenges to adopting stronger consistency at scale. In *HotOS XV*, 2015.

[17] P. Bailis, A. Davidson, A. Fekete, A. Ghodsi, J. M. Hellerstein, and I. Stoica. Highly available transactions: Virtues and limitations. *VLDB*, 7(3), 2013.

[18] P. Bailis, A. Fekete, M. J. Franklin, A. Ghodsi, J. M. Hellerstein, and I. Stoica. Feral Concurrency Control: An Empirical Investigation of Modern Application Integrity. In *SIGMOD*, 2015.

[19] P. Bailis, S. Venkataraman, M. J. Franklin, J. M. Hellerstein, and I. Stoica. Probabilistically bounded staleness for practical partial quorums. *VLDB*, 5(8), 2012.

[20] J. Baker, C. Bond, J. C. Corbett, J. Furman, A. Khorlin, J. Larson, J.-M. Leon, Y. Li, A. Lloyd, and V. Yushprakh. Megastore: Providing scalable, highly available storage for interactive services. In *CIDR*, 2011.

[21] E. Brewer. Cap twelve years later: How the ”rules” have changed. *Computer*, 45(2), 2012.

[22] M. Burrows. The Chubby lock service for loosely-coupled distributed systems. In *OSDI*, 2006.

[23] T. D. Chandra, R. Griesemer, and J. Redstone. Paxos made live: An engineering perspective. In *PODC*, 2007.

[24] 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*, 1(2), 2008.

[25] B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears. Benchmarking cloud serving systems with YCSB. In *SoCC*, 2010.

[26] J. C. Corbett, J. Dean, M. Epstein, A. Fikes, C. Frost, J. 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. *ACM Transactions on Computer Systems (TOCS)*, 31(3), 2013.

[27] J. Dean and L. A. Barroso. The tail at scale. *CACM*, 56(2), 2013.

[28] G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, and W. Vogels. Dynamo. In *SOSP*, 2007.

[29] A. Dragojević, D. Narayanan, E. B. Nightingale, M. Renzelmann, A. Shamis, A. Badam, and M. Castro. No Compromises: Distributed Transactions with Consistency, Availability, and Performance. In *SOSP*, 2015.

[30] J. Du, C. Iorgulescu, A. Roy, and W. Zwaenepoel. Gentlerain: Cheap and scalable causal consistency with physical clocks. In *SoCC*, 2014.

[31] M. Eriksen. Your server as a function. In *PLOS*, 2013.

[32] D. K. Gifford. Weighted voting for replicated data. In *SOSP*, 1979.

[33] S. Gilbert and N. Lynch. Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services. *ACM SIGACT News*, 33(2), 2002.

[34] C. Hale and R. Kennedy. Using Riak at Yammer, March 2011. http://dl.dropbox.com/u/2744222/2011-03-22_Riak-At-Yammer.pdf.

[35] J. Hamilton. The cost of latency. http://perspectives.mvdirona.com/2009/10/the-cost-of-latency/, 2009.

[36] P. Helland and D. Campbell. Building on quicksand. In *CIDR*, 2009.

[37] M. Herlihy. Wait-free synchronization. *ACM Trans. Program. Lang. Syst.*, 1991.

[38] M. P. Herlihy and J. M. Wing. Linearizability: A correctness condition for concurrent objects. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 12(3), 1990.

[39] P. Hunt, M. Konar, F. P. Junqueira, and B. Reed. Zookeeper: Wait-free coordination for internet-scale systems. In *USENIX ATC*, 2010.

[40] F. P. Junqueira, B. C. Reed, and M. Serafini. Zab: High-performance broadcast for primary-backup systems. In *DSN*, 2011.

[41] M. Kapritsos, Y. Wang, V. Quema, A. Clement, L. Alvisi, and M. Dahlin. All about Eve: Execute-Verify Replication for Multi-Core Servers. In *OSDI*, 2012.

[42] Y. Koren. Collaborative filtering with temporal dynamics. *CACM*, 53(4), 2010.

[43] T. Kraska, M. Hentschel, G. Alonso, and D. Kossmann. Consistency Rationing in the Cloud: Pay only when it matters. *VLDB*, 2(1), 2009.

[44] T. Kraska, G. Pang, M. J. Franklin, S. Madden, and A. Fekete. MDCC: Multi-data Center Consistency. In *EuroSys*, 2013.

[45] A. Lakshman and P. Malik. Cassandra - A Decentralized Structured Storage System. *SIGOPS Oper. Syst. Rev.*, 44(2), 2010.

[46] L. Lamport. The part-time parliament. *ACM TOCS*, 16(2), 1998.

[47] J. R. Lange, P. A. Dinda, and S. Rossoff. Experiences with Client-based Speculative Remote Display. In *USENIX ATC*, 2008.

[48] C. Lee, S. J. Park, A. Kejriwal, S. Matsushita, and J. Ousterhout. Implementing Linearizability at Large Scale and Low Latency. In *SOSP*, 2015.

[49] K. Lee, D. Chu, E. Cuervo, J. Kopf, Y. Degtyarev, S. Grizan, A. Wolman, and J. Flinn. Outatime: Using speculation to enable low-latency continuous interaction for mobile cloud gaming. In *MobiSys*, 2015.

[50] C. Li, J. Leitao, A. Clement, N. Preguiça, R. Rodrigues, and V. Vafeiadis. Automating the choice of consistency levels in replicated systems. In *USENIX ATC*, 2014.

[51] C. Li, D. Porto, A. Clement, J. Gehrke, N. Preguiça, and R. Rodrigues. Making geo-replicated systems fast as possible, consistent when necessary. In *OSDI*, 2012.

[52] D. Li, J. Mickens, S. Nath, and L. Ravindranath. Domino: Understanding Wide-Area, Asynchronous Event Causality in Web Applications. In *SoCC*, 2015.

[53] B. Liskov and L. Shrira. Promises: Linguistic support for efficient asynchronous procedure calls in distributed systems. In *PLDI*, 1988.

[54] W. Lloyd, M. J. Freedman, M. Kaminsky, and D. G. Andersen. Stronger semantics for low-latency geo-replicated storage. In *NSDI*, 2013.

[55] H. Lu, K. Veeraraghavan, P. Ajoux, J. Hunt, Y. J. Song, W. Tobagus, S. Kumar, and W. Lloyd. Existential Consistency: Measuring and Understanding Consistency at Facebook. In *SOSP*, 2015.

[56] E. Meijer. Your mouse is a database. *CACM*, 55(5), 2012.

[57] J. Mickens, J. Elson, J. Howell, and J. Lorch. Crom: Faster Web Browsing Using Speculative Execution. In *NSDI*, 2010.

[58] S. Nakamoto. Bitcoin: A peer-to-peer electronic cash system.

[59] E. B. Nightingale, P. M. Chen, and J. Flinn. Speculative Execution in a Distributed File System. In *SOSP*, 2005.

[60] R. Nishtala, H. Fugal, S. Grimm, M. Kwiatkowski, H. Lee, H. C. Li, R. McElroy, M. Paleczny, D. Peek, P. Saab, et al. Scaling Memcache at Facebook. In *NSDI*, 2013.

[61] G. Pang, T. Kraska, M. J. Franklin, and A. Fekete. PLANET: Making Progress with Commit Processing in Unpredictable Environments. In *SIGMOD*, 2014.

[62] D. Perkins, N. Agrawal, A. Aranya, C. Yu, Y. Go, H. V. Madhyastha, and C. Ungureanu. Simba: Tunable End-to-end Data Consistency for Mobile Apps. In *EuroSys*, 2015.

[63] Y. Saito and M. Shapiro. Optimistic replication. *ACM Computing Surveys*, 37(1), 2005.

[64] P. Schulle. Manhattan, distributed database for Twitter scale. http://tiny.cc/twitmanhattan, 2014.

[65] M. Schwarzkopf. *Operating system support for warehouse-scale computing*. PhD thesis, University of Cambridge Computer Laboratory, 2015.

[66] D. B. Terry, V. Prabhakaran, R. Kotla, M. Balakrishnan, M. K. Aguilera, and H. Abu-Libdeh. Consistency-based service level agreements for cloud storage. In *SOSP*, 2013.

[67] D. B. Terry, M. M. Theimer, K. Petersen, a. J. Demers, M. J. Spreitzer, and C. H. Hauser. Managing update conflicts in Bayou, a weakly connected replicated storage system. *SIGOPS Oper. Syst. Rev.*, 29(5), 1995.

[68] R. Van Renesse and F. B. Schneider. Chain replication for supporting high throughput and availability. In *OSDI*, 2004.

[69] H. Wada, A. Fekete, L. Zhao, K. Lee, and A. Liu. Data Consistency Properties and the Trade-offs in Commercial Cloud Storage: the Consumers' Perspective. In *CIDR*, 2011.

[70] B. Wester, P. M. Chen, and J. Flinn. Operating System Support for Application-Specific Speculation. In *EuroSys*, 2011.

[71] B. Wester, J. A. Cowling, E. B. Nightingale, P. M. Chen, J. Flinn, and B. Liskov. Tolerating latency in replicated state machines through client speculation. In *NSDI*, 2009.

[72] C. Xie, C. Su, M. Kapritsos, Y. Wang, N. Yaghmazadeh, L. Alvisi, and P. Mahajan. Salt: Combining ACID and BASE in a distributed database. In *OSDI*, 2014.

[73] H. Yu and A. Vahdat. Design and evaluation of a continuous consistency model for replicated services. In *OSDI*, 2000.

[74] I. Zhang, N. K. Sharma, A. Szekeres, A. Krishnamurthy, and D. R. K. Ports. Building Consistent Transactions with Inconsistent Replication. In *SOSP*, 2015.

FaSST: Fast, Scalable and Simple Distributed Transactions with Two-sided (RDMA) Datagram RPCs

Anuj Kalia Michael Kaminsky[†] David G. Andersen
Carnegie Mellon University [†]*Intel Labs*

Abstract

FaSST is an RDMA-based system that provides distributed in-memory transactions with serializability and durability. Existing RDMA-based transaction processing systems use one-sided RDMA primitives for their ability to bypass the remote CPU. This design choice brings several drawbacks. First, the limited flexibility of one-sided RDMA reduces performance and increases software complexity when designing distributed data stores. Second, deep-rooted technical limitations of RDMA hardware limit scalability in large clusters. FaSST eschews one-sided RDMA for fast RPCs using two-sided unreliable datagrams, which we show drop packets extremely rarely on modern RDMA networks. This approach provides better performance, scalability, and simplicity, without requiring expensive reliability mechanisms in software. In comparison with published numbers, FaSST outperforms FaRM on the TATP benchmark by almost 2x while using close to half the hardware resources, and it outperforms DrTM+R on the SmallBank benchmark by around 1.7x without making data locality assumptions.

1 Introduction

"Remote procedure calls (RPC) appear to be a useful paradigm."
— Birrel & Nelson, 1984

Serializable distributed transactions provide a powerful programming abstraction for designing distributed systems such as object stores and on-line transaction processing (OLTP) systems. Although earlier work in this space sacrificed strong transactional semantics for performance [9], recent systems have shown that transactions can be fast in the datacenter [12, 28, 7, 5].[1] The key enablers are high-speed networks and lightweight network stacks (i.e., kernel bypass). In addition, these systems exploit Remote Direct Memory Access (RDMA) for its low latency and CPU efficiency. A common thread in these systems is that they make extensive use of one-sided RDMA operations that bypass the remote CPU. The intent behind this decision is to harness one-sided RDMA's ability to save remote CPU cycles.

[1]We discuss only distributed transactions in this paper, so we use the more general but shorter term transactions.

In this paper, we explore whether one-sided RDMA is not the best choice for designing transaction processing systems. First, there is a gap between the paradigm of one-sided RDMA, and capabilities needed for efficient transactional access to remote data stores. One-sided RDMA provides only remote reads, writes, and atomic operations, whereas accessing data stores typically involves traversing data structures such as hash tables and B-Trees. In general, these structures consist of an index for fast lookup, and the actual data, requiring two or more RDMA reads to access data. This leads to lower throughput and higher latency, and reduces the net CPU savings from remote CPU bypass [15]. The key technique to overcome this gap is to *flatten* the data structure, by either ignoring the index [5], merging the data with the index [12], or caching the index [12, 28, 7] at all servers. Each of these variants has an associated cost in generality and system performance. Second, the connection-oriented nature of current one-sided RDMA implementations typically requires CPU cores to share local NIC queue pairs for scalability [11], reducing local per-core RDMA throughput by several factors, and the net benefit of remote CPU bypass.

We show that there is a better primitive for transactions: remote procedure calls (RPCs) over two-sided unreliable datagram messages. RPCs involve the remote CPU in message processing and are more flexible than one-sided RDMA, allowing data access in a single round trip [15]. However, previous RPC implementations over RDMA either performed poorly (e.g., up to 4x worse than one-sided RDMA in FaRM [12]), or were specialized (e.g., HERD's RPCs [15] deliver high performance for all-to-one communication where one server handles RPCs from many clients). A key contribution of this work is FaSST RPCs: an all-to-all RPC system that is fast, scalable, and CPU-efficient. This is made possible by using RDMA's datagram transport that provides scalability, and allows "Doorbell batching" which saves CPU cycles by reducing CPU-initiated PCIe bus transactions. We show that FaSST RPCs provide (1) up to 8x higher throughput, and 13.9x higher CPU efficiency than FaRM's RPCs (Section 4.5), and (2) 1.7–2.15x higher CPU efficiency, or higher throughput, than one-sided READs, depending on whether or not the READs scale to clusters with more than a few tens of nodes (Section 3.3).

Using an unreliable transport layer requires handling packet loss. In RDMA networks such as InfiniBand, however, packet loss is extremely rare because the underlying link layer provides reliability. We did not observe any lost packets in our experiments that transmitted over 50 PB of network data on a real-world InfiniBand cluster with up to 69 nodes. Nevertheless, packet loss can occur during hardware failures, and corner cases of the link-layer's reliability protocol. We detect these losses using coarse-grained timeouts triggered at the RPC requester, and describe how they can be handled similarly to conventional machine failures.

FaSST is a new transaction processing system built on FaSST RPCs. It uses optimistic concurrency control, two-phase commit, and primary-backup replication. Our current implementation supports transactions on an unordered key-value store based on MICA [18], and maps 8-byte keys to opaque objects. We evaluate FaSST using three workloads: a transactional object store, a read-mostly OLTP benchmark called TATP, and a write-intensive OLTP benchmark called SmallBank. FaSST compares favorably against published per-machine throughput numbers. On TATP, FaSST outperforms FaRM [12] by 1.87x when using close to half the hardware (NIC and CPU) resources. On SmallBank, FaSST outperforms DrTM+R [7] by 1.68x with similar hardware without making data locality assumptions. The source code for FaSST and the experiments in this paper is available at `https://github.com/efficient/fasst`.

2 Background

2.1 Fast distributed transactions

This section outlines the environment that we target with FaSST. FaSST aims to provide distributed transactions inside a single datacenter where a single instance of the system can scale to a few hundred nodes. Each node in the system is responsible for a partition of the data based on a primary key, and nodes operate in the *symmetric model*, whereby each node acts both as a client and a server. For workloads with good data locality (e.g., transactions that only access data in one partition), the symmetric model can achieve higher performance by co-locating transactions with the data they access [11, 12].

FaSST targets high-speed, low-latency key-value transaction processing with throughputs of several million transactions/sec and average latencies around one hundred microseconds on common OLTP benchmarks with short transactions with up to a few tens of keys. Achieving this performance requires in-memory transaction processing, and fast userspace network I/O with polling (i.e., the overhead of a kernel network stack or interrupts is unacceptable). We assume commercially available network equipment: 10-100 Gbps of per-port bandwidth and $\approx$ 2 µs end-to-end latency.

Making data durable across machine failures requires logging transactions to persistent storage, and quick recovery requires maintaining multiple replicas of the data store. Keeping persistent storage such as disk or SSDs on the critical path of transactions limits performance. Similar to recent work, FaSST assumes that the transaction processing nodes are equipped with battery-backed DRAM [12], though future NVRAM technologies, if fast enough, would also work.

Finally, FaSST uses primary-backup replication to achieve fault tolerance. We assume that failures will be handled using a separate fault-tolerant configuration manager that is off of the critical path (the Vertical Paxos model [17]), similar to recent work on RDMA-based distributed transactions [12, 7]. We do not currently implement such a configuration manager.

2.2 RDMA

RDMA is a networking concept of which several implementations exist. The Virtual Interface Architecture (VIA) is a popular model for user-level, zero-copy networking [13], and forms the basis of current commodity RDMA implementations such as InfiniBand, RoCE (RDMA over Converged Ethernet), and iWARP (internet Wide Area RDMA Protocol). VIA NICs provide user processes with virtual interfaces to the network. VIA is fundamentally connection-oriented: a connection must be established between a pair of virtual interfaces before they are allowed to communicate. This design decision was made by VIA architects to simplify VIA implementations and reduce latency [13]. The discussion in this paper, and some of our contributions are specific to VIA-based RDMA implementations; we discuss other, non-commodity RDMA implementations in Section 7.1.

In VIA-based RDMA implementations, virtual interfaces are called queue pairs (QPs), each consisting of a send queue and a receive queue. Processes access QPs by posting *verbs* to these queues. Two-sided verbs—SEND and RECV—require involvement of the CPU at both the sender and receiver: a SEND generates a message whose data is written to a buffer specified by the receiver in a pre-posted RECV. One-sided verbs—READ, WRITE, and ATOMIC—bypass the remote CPU to operate directly on remote memory.

RDMA transports can be connected or connectionless. Connected transports offer one-to-one communication between two queue pairs: to communicate with N remote machines, a thread must create N QPs. These transports provide one-sided RDMA and end-to-end reliability, but do not scale well to large clusters. This is because NICs have limited memory to cache QP state, and exceeding

	SEND/RECV	WRITE	READ/ATOMIC
RC	✓	✓	✓
UC	✓	✓	✗
UD	✓	✗	✗

Table 1: Verbs supported by each transport type. RC, UC, and UD stand for Reliable Connected, Unreliable Connected, and Unreliable Datagram, respectively.

the size of this state by using too many QPs causes cache thrashing [11]. Connectionless (datagram) transports are extensions to the connection-oriented VIA, and support fewer features than connected transports: they do not provide one-sided RDMA or end-to-end reliability. However, they allow a QP to communicate with multiple other QPs, and have better scalability than connected transports as only one QP is needed per thread.

RDMA transports can further be either reliable or unreliable, although current commodity NICs do not provide a reliable datagram transport. Reliable transports provide in-order delivery of messages and return an error in case of failure. Unreliable transports achieve higher performance by avoiding acknowledgment packets, but do not provide reliability guarantees or return an error on network failures. Modern high-speed networks, including Mellanox's InfiniBand and Intel's OmniPath, also provide reliability below the transport layer [3, 6]. Their link layer uses flow control to prevent congestion-based losses, and retransmissions to prevent bit error-based losses. InfiniBand's physical layer uses Forward Error Correction to fix most bit errors, which themselves are rare. For example, the bit error rate of the InfiniBand cables used in our clusters is less than 10^{-15}. Therefore even unreliable transports, which lack end-to-end reliability, lose packets extremely rarely: we did not lose any packets in around 50 PB of unreliable data transfer (Section 3.4). Note that link-layer flow control in these networks can cause congestion collapse in rare scenarios, leading to low throughput, but not dropped packets.

Current RDMA implementations provide three main transports: Reliable Connected (RC), Unreliable Connected (UC), and Unreliable Datagram (UD). Table 1 shows the subset of verbs supported by implementations of each transport. Not all transport layers provide all types of verbs, so choosing a verb means accepting the limitations of the available transports. Note that only connected transports provide one-sided verbs, limiting the scalability of designs that use these verbs.

3 Choosing networking primitives

We now describe the rationale behind our decision to build an RPC layer using two-sided datagram verbs. We show that RPCs are:

Name	**Hardware**
CX3	Mellanox ConnectX-3 (1x 56 Gb/s InfiniBand ports), PCIe 3.0 x8, Intel® Xeon® E5-2450 CPU (8 cores, 2.1 GHz), 16 GB DRAM
CIB	Mellanox Connect-IB (2x 56 Gb/s InfiniBand ports), PCIe 3.0 x16, Intel® Xeon® E5-2683-v3 CPU (14 cores, 2 GHz), 192 GB DRAM

Table 2: Measurement clusters

1. **Fast**: Although READs can outperform similarly-sized RPCs on small clusters, RPCs perform better when accounting for the amplification in size or number of READs required to access real data stores.
2. **Scalable**: Datagram RPC throughput and CPU use remains stable as the cluster size increases, whereas READ performance degrades because READs must use connected transport with today's NICs.
3. **Simple**: RPCs reduce the software complexity required to design distributed data stores and transactions compared to one-sided RDMA-based systems.

3.1 Advantage of RPCs

Recent work on designing distributed data stores over RDMA-capable networks has largely focused on how to use one-sided RDMA primitives. In these designs, clients access remote data structures in servers' memory using one or more READs, similar to how one would access data in local memory. Various optimizations help reduce the number of READs needed; we discuss two such optimizations and their limitations below.

Value-in-index: FaRM [11, 12] provides hash table access in $\approx$ 1 READ on average by using a specialized index that stores data adjacent to its index entry, allowing data to be READ with the index. However, doing so amplifies the size of the READ by a factor of 6–8x, reducing throughput [15]. This result highlights the importance of comparing the application-level capabilities of networking primitives: although micro-benchmarks suggest that READs can outperform similar-sized RPCs, READs require extra network traffic and/or round-trips due to their one-sided nature, tipping the scales in the other direction.

Caching the index: DrTM [28, 7] caches the index of its hash table at all servers in the cluster, allowing single-READ GETs; FaRM [12] uses a similar approach for its B-Tree. Although this approach works well when the workload has high locality or skew, it does not work in general because indexes can be large: Zhang et al. report that indexes occupy over 35% of memory for popular OLTP benchmarks in a single-node transaction processing system [30]; the percentage is similar in our implementation of distributed transaction processing benchmarks. In this case, caching even 10% of the index requires *each* machine to dedicate 3.5% of the *total cluster memory* *capacity* for the index, which is impossible if the cluster contains more than $100/3.5 \approx 29$ nodes. The Cell B-Tree [21] caches B-Tree nodes 4 levels above the leaf nodes to save memory and reduce churn, but requires multiple round trips (~ 4) when clients access the B-Tree using READs.

RPCs allow access to partitioned data stores with two messages: the request and the reply. They do not require message size amplification, multiple round trips, or caching. The simplicity of RPC-based programming reduces the software complexity required to take advantage of modern fast networks in transaction processing: to implement a partitioned, distributed data store, the user writes only short RPC handlers for a single-node data store. This approach eliminates the software complexity required for one-sided RDMA-based approaches [11, 21]. For example, in this paper, we use MICA's hash table design [18] for unordered key-value storage. We made only minor modifications to the MICA codebase to support distributed transactions. In the future, we plan to use Masstree [19] for ordered storage.

3.2 Advantage of datagram transport

Datagram transport allows each CPU core to create one datagram QP that can communicate with all remote cores. Since the number of QPs is relatively small (as many as the number of cores), providing each core exclusive access to QPs is possible without overflowing the NIC's cache. Providing exclusive access to QPs with connected transport, however, is not scalable: In a cluster with N machines and T threads per machine, doing so requires $N*T$ QPs at every machine, which may not fit in the NIC's queue pair cache. Threads can share QPs to reduce the QP memory footprint [11]. Sharing QPs reduces CPU efficiency because threads contend for locks, and the cache lines for QP buffers bounce between their CPU cores. The effect can be dramatic: in our experiments, QP sharing reduces the per-core throughput of one-sided READs by up to 5.4x (Section 3.3.2). Similarly, FaRM's RPCs that use one-sided WRITEs and QP sharing become CPU-bottlenecked at 5 million requests/sec (Mrps) per machine [12]. Our datagram-based RPCs, however, do not require QP sharing and achieve up to 40.9 Mrps/machine, and even then they are bottlenecked by the NIC, not CPU (Section 3.3.1).

In comparison with connected transports, datagram transport confers a second important advantage in addition to scalability: *Doorbell batching* reduces CPU use. We describe this feature in a simplified form here; a detailed discussion is available in our earlier paper [16]. User processes post operations to the NIC by writing to a per-QP Doorbell register on the NIC over the PCIe bus, specifying the number of new operations on that QP. This write is relatively expensive for the CPU because it requires flushing the write buffers, and using memory barriers for ordering. In transactional systems, however, applications can amortize this cost by issuing multiple RDMA work requests at a time. Examples include reading or validating multiple keys for multi-key transactions, or sending update messages to the replicas of a key. With a datagram QP, the process only needs to ring the Doorbell once per batch, regardless of the individual message destinations within the batch. With connected QPs, however, the process must ring multiple Doorbells—as many as the number of destinations appearing in the batch. Note that Doorbell batching does not coalesce packets at the RDMA layer (i.e., it does not put multiple application-level requests in a single RDMA packet); Doorbell batching also does not add latency because we do it opportunistically, i.e., we do not wait for a batch of messages to accumulate.

3.3 Performance considerations

Two recent projects study the relative performance of RPCs and one-sided RDMA. In the asymmetric setting where multiple clients send requests to one server, HERD shows that RPCs perform similarly to READs [15]. In HERD, clients send requests to the server via WRITEs over UC; the server responds with SENDs over UD. This approach scales well with the number of clients because the number of *active* queue pairs at the server is small. The server's UC QPs are passive because the server's CPU does not access them; these passive QPs consume little memory in the NIC. The active UD QPs are few in number.

Unfortunately, as noted by Dragojevic et al. [12], HERD's RPC design does not scale well in the symmetric setting required for distributed transactions, where every machine issues requests and responses. This scenario requires many active UC QPs on each node for sending requests. In FaRM's experiments [12] in the symmetric setting, READs outperform *their* RPCs by 4x.

We now present experimental results showing that FaSST's RPCs are a better choice than one-sided RDMA for distributed transactions. The design and implementation of our RPC system is discussed in detail in Section 4; here, we use it to implement basic RPCs where both the request and reply are fixed-size buffers. We first compare the raw throughput of RPCs and one-sided READs by using a small cluster where READs do not require QP sharing. Next, we compare their performance on more realistic, medium-sized clusters.

Clusters used: To show that our results generalize to a range of RDMA hardware, we use two clusters with different NICs and CPU processing power (Table 2). The clusters are named after the initials of their NIC. CX3 is a shared Emulab [29] cluster with 192 nodes; our experiments used up to 69 nodes, depending on the availability of nodes. CX3 nodes have a ConnectX-3 NIC and an

Intel SandyBridge CPU with 8 cores. CIB is a private cluster with 11 nodes. CIB nodes have a more powerful Connect-IB NIC that provides 2x more bandwidth and around 4x higher message rate than a ConnectX-3 NIC. They also have a more powerful, 14-core Intel Haswell CPU.

Experiment setup: We use a cluster of machines in a symmetric setting, i.e., every machine issues requests (RPC requests or READs) to every other machine. For READs without QP sharing, each thread creates as many RC QPs as the number of machines, and issues READs to randomly chosen machines. We evaluate RPC performance for two request batch sizes (1 and 11) to show the effect of Doorbell batching for requests. We prevent RPC request coalescing (Section 4) by sending each request in a batch to a different machine; this restricts our maximum batch size on CIB to 11.

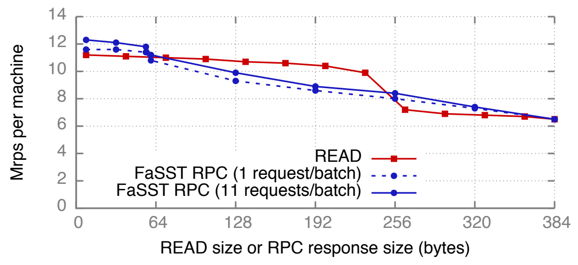
We compare RPC and READ performance for different response sizes; for RPCs, the request size is fixed at 32 bytes, which is sufficient to read from FaSST's data stores. We report millions of requests per second per machine (Mrps/machine). Note that for RPCs, each machine's CPU also serves responses to requests from other machines, so the number of messages sent by a machine is approximately twice the request rate that we report. Our results show that:

1. FaSST RPCs provide good raw throughput. For small messages up to 56 bytes, RPCs deliver a significant percentage of the maximum throughput of similar-sized READs on small clusters: 103–106% on CX3 and 68–80% on CIB, depending on the request batch size. When accounting for the amplification in READ size or number required to access data structures in real data stores, RPCs deliver higher raw throughput than READs.
2. On medium-sized clusters, if READs do not share QPs, RPCs provide 1.38x and 10.1x higher throughput on CIB and CX3, respectively. If READs do share QPs, their CPU efficiency drops by up to 5.4x, and RPCs provide 1.7–2.15x higher CPU efficiency.

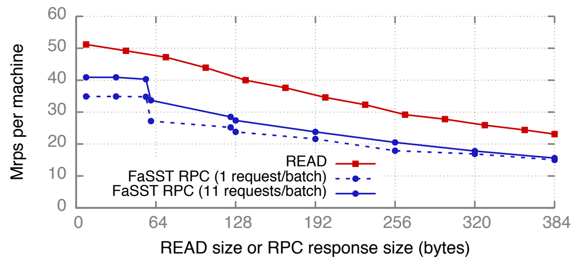
These experiments highlight the sometimes dramatic difference in performance between micro-benchmarks and more realistic settings.

3.3.1 On small clusters

To measure the maximum raw throughput of READs, we use 6 nodes so that only a small number of QPs are needed even for READs: each node on CX3 (8 cores) and CIB (14 cores) uses 48 and 84 QPs, respectively. We use 11 nodes for RPCs to measure performance with a large request batch size—using only 6 nodes for RPCs would restrict the maximum non-coalesced request batch size to 6. (As shown in Section 3.3.2, using 11 nodes for READs gives lower throughput due to cache misses in the NIC, so we use fewer nodes to measure their peak throughput.) Figure 1 shows Mrps/machine for READs and RPCs on the two clusters.



(a) CX3 cluster (ConnectX-3 NIC)



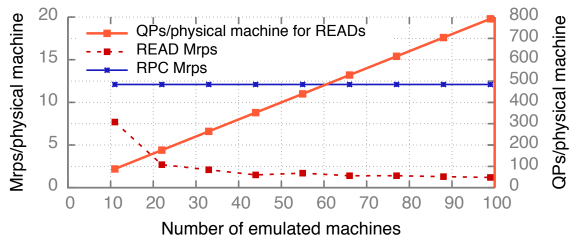
(b) CIB cluster (Connect-IB NIC)

Figure 1: Small clusters: Throughput comparison of FaSST RPCs (11 nodes) and READs (6 nodes). Note that the two graphs use a different Y scale.

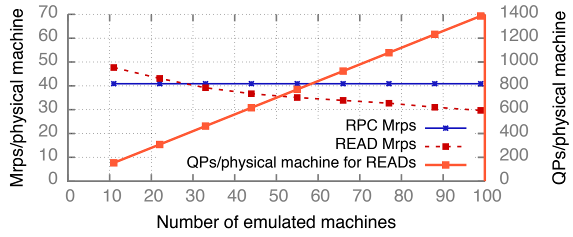
Raw throughput: Depending on the request batch size, FaSST RPCs deliver up to 11.6–12.3 Mrps on CX3, and 34.9–40.9 Mrps on CIB. READs deliver up to 11.2 Mrps on CX3, and 51.2 Mrps on CIB. The throughput of both RPCs and READs is bottlenecked by the NIC: although our experiment used all cores on both clusters, fewer cores can achieve similar throughput, indicating that the CPU is not the bottleneck.

Comparison with READs: Although RPCs usually deliver lower throughput than READs, the difference is small. For response sizes up to 56 bytes, which are common in OLTP, RPC throughput is within 103–106% of READ throughput on CX3, and 68–80% of READ throughput on CIB, depending on the request batch size. For larger responses, READs usually outperform our RPCs, but the difference is smaller than 4x, as is the case for FaRM's one-sided RPCs. This is because FaSST's RPCs are bottlenecked by the NIC on both clusters, whereas FaRM's RPCs become CPU-bottlenecked due to QP sharing (Section 3.3.2). As noted above, these "raw" results are only baseline micro-benchmarks; the following paragraphs consider the numbers in the context of "real-world" settings.

Effect of multiple READs: In all cases (i.e., regardless of cluster used, response size, and request batch size), RPCs provide higher throughput than using 2 READs.



(a) CX3 cluster (ConnectX-3 NIC)



(b) CIB cluster (Connect-IB NIC)

Figure 2: Comparison of FaSST RPC and READ throughput, and the number of QPs used for READs with increasing *emulated* cluster size.

Thus, for any data store/data structure that requires two or more READs, RPCs provide strictly higher throughput.

Effect of larger READs: Consider, for example, a hash table that maps 8-byte keys to 40-byte values (this configuration is used in one of the database tables in the TATP benchmark in Section 6) on CX3. For this hash table, FaRM's single-READ GETs require approximately 384-byte READs (8x amplification) and can achieve up to 6.5 Mrps/machine on CX3. With FaSST RPCs, these key-value requests can be handled in one RPC with an 8-byte request and a 40-byte response (excluding header overheads), and can achieve 11.4–11.8 Mrps/machine (over 75% higher) before the ConnectX-3 NIC becomes the bottleneck. On CIB, 384-byte READs achieve 23.1 Mrps, whereas FaSST RPCs achieve 34.9–40.9 Mrps (over 51% higher).

3.3.2 On medium-sized clusters

Measuring the impact of one-sided RDMA's poor scalability requires more nodes. As the CIB cluster has only 11 physical machines, we emulate the effect of a larger cluster by creating as many QPs on each machine as would be used in the larger cluster. With N physical nodes, we emulate clusters of $N * M$ nodes for different values of M. Instead of creating N QPs, each worker thread creates $N * M$ QPs, and connects them to QPs on other nodes. Note that we only do so for READs because for FaSST's RPCs, the number of local QPs does not depend on the number of machines in the cluster.

Figure 2 compares READ and RPC throughput for increasing emulated cluster sizes. We use 32-byte READs and RPC requests and responses. Note that the peak READ throughput in this graph is lower than Figure 1 that used 6 nodes. This is because NIC cache misses occur with as few as 11 nodes. On CX3, READ throughput drops to 24% of its peak with as few as 22 emulated nodes. On CIB, READs lose their throughput advantage over RPCs on clusters with 33 or more nodes. The decline with Connect-IB NICs is more gradual than with ConnectX-3 NICs. This may be due to a larger cache or better cache miss pipelining [10] in the Connect-IB NIC. Section 7.2 discusses the possible impact of future NIC and CPU hardware on queue pair scalability.

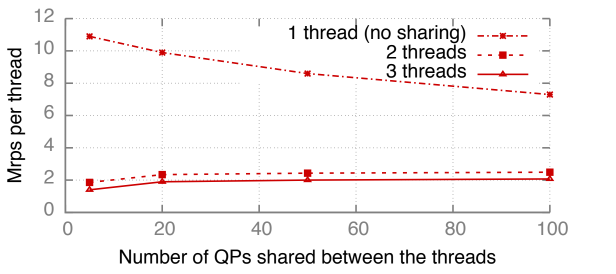


Figure 3: Per-thread READ throughput with QP sharing (CIB)

Sharing QPs: Fewer QPs are required if they are shared between worker threads, but doing so drastically reduces the CPU efficiency of one-sided RDMA. QP sharing is typically implemented by creating several sets of N QPs, where each set is connected to the N machines [11]. A machine's threads are also grouped into sets, and threads in a set share a QP set.

We measure the loss in CPU efficiency as follows. We use one server machine that creates a tuneable number of QPs and connects them to QPs spread across 5 client machines (this is large enough to prevent the clients from becoming a bottleneck). We run a tuneable number of worker threads on the server that share these QPs, issuing READs on QPs chosen uniformly at random.

We choose the number of QPs and threads per set based on a large hypothetical cluster with 100 nodes and CIB's CPUs and NICs. A Connect-IB NIC supports $\approx$ 400 QPs before READ throughput drops below RPC throughput (Figure 2). In this 100-node cluster, the 400 QPs are used to create 4 sets of 100 connections (QPs) to remote machines. CIB's CPUs have 14 cores, so sets of 3–4 threads share a QP set.

Figure 3 shows per-thread throughput in this experiment. For brevity, we only show results on CIB; the loss in CPU efficiency is comparable on CX3. The hypothetical configuration above requires sharing 100 QPs among at least 3 threads; we also show other configurations that may be relevant for other NICs and cluster sizes. With one thread, there is no sharing of QPs and throughput is high—up to 10.9 Mrps. Throughput with QP sharing between 3 threads, however, is 5.4x lower (2 Mrps).

This observation leads to an important question: If the increase in CPU utilization at the local CPU due to QP sharing is accounted for, do one-sided READs use fewer *cluster-wide* CPU cycles than FaSST's RPCs that do not require QP sharing? We show in Section 4 that the answer is no. FaSST's RPCs provide 3.4–4.3 Mrps per core on CIB—1.7–2.15x higher than READs with QP sharing between 3 threads. Note that in our symmetric setting, each core runs both client and server code. Therefore, READs use cluster CPU cycles at only the client, whereas RPCs use them at both the client and the server. However, RPCs consume fewer *overall* CPU cycles.

3.4 Reliability considerations

Unreliable transports do not provide reliable packet delivery, which can introduce programming complexity and/or have performance implications (e.g., increased CPU use), since reliability mechanisms such as timeouts and retransmissions must be implemented in the software RPC layer or application.

To understand FaSST's approach to handling potential packet loss, we make two observations. First, we note that transaction processing systems usually include a reconfiguration mechanism to handle node failures. Reconfiguration includes optionally pausing ongoing transactions, informing nodes of the new cluster membership, replaying transaction logs, and re-replicating lost data [12]. In FaSST, we assume a standard reconfiguration mechanism; we have not implemented such a mechanism because this paper's contribution is not in that space. We expect that, similar to DrTM+R [7], FaRM's recovery protocol [12] can be adapted to FaSST.

The second observation is that in normal operation, packet loss in modern RDMA-capable networks is extremely rare: in our experiments (discussed below), we observed zero packet loss in over 50 PB of data transferred. Packets can be lost during network hardware failures, and corner cases of the link/physical layer reliability protocols. FaSST's RPC layer detects these losses using coarse-grained timeouts maintained by the RPC requester (Section 4.3).

Based on these two observations, we believe that an acceptable first solution for handling packet loss in FaSST is to simply restart one of the two FaSST processes that is affected by the lost RPC packet, allowing the reconfiguration mechanism to make the commit decision for the affected transaction. We discuss this in more detail in Section 5.1.

3.4.1 Stress tests for packet loss

Restarting a process on packet loss requires packet losses to be extremely rare. To quantify packet loss on realistic RDMA networks, we set up an experiment on the CX3 cluster, which is similar to real-world clusters with multiple switches, oversubscription, and sharing. It is a shared cluster with 192 machines arranged in a tree topology with seven leaf and two spine switches, with an oversubscription ratio of 3.5. The network is shared by Emulab users. Our largest experiment used 69 machines connected to five leaf switches.

Threads on these machines use UD transport to exchange 256-byte RPCs. We used 256-byte messages to achieve both high network utilization and message rate. Threads send 16 requests to remote threads chosen uniformly at random, and wait for all responses to arrive before starting the next batch. A thread stops making progress if a request or reply packet is lost. Threads routinely output their progress messages to a log file; we manually inspect these files to ensure that all threads are making progress.

We ran the experiment without a packet loss for approximately 46 hours. (We stopped the experiment when a log file exhausted a node's disk capacity.) The experiment generated around 100 trillion RPC packets and 33.2 PB of network data. Including other smaller-scale experiments with 20–22 nodes, we have transferred over 50 PB of network data without a lost packet.

While we observed zero packet loss, we detected several reordered packets. Using sequence numbers embedded in the RPC packets, we observed around 1500 reordered packets in 100 trillion packets transferred. Reordering happens due to multi-path in CX3: although there is usually a single deterministic path between each source and destination node, the InfiniBand subnet manager sometimes reconfigures the switch routing tables to use different paths.

4 FaSST RPCs

FaSST's RPCs are designed for transaction workloads that use small (~ 100 byte) objects and a few tens of keys. This layer abstracts away details of RDMA, and is used by higher-layer systems such as our transaction processing system. Key features of FaSST's RPCs include integration with coroutines for efficient network latency hiding, and optimizations such as Doorbell batching and message coalescing.

4.1 Coroutines

RDMA network latency is on the order of 10 µs under load, which is much higher than the time spent by our applications in computation and local data store accesses. It is critical to not block a thread while waiting for an RPC reply. Similar to Grappa [22], FaSST uses coroutines (cooperative multitasking) to hide network latency: a coroutine yields after initiating network I/O, allowing other coroutines to do useful work while the RPC is in flight. Our experiments showed that a small number (~ 20) of coroutines per thread is sufficient for latency hiding, so FaSST uses standard coroutines from the Boost C++ library instead of Grappa's coroutines, which are optimized for use cases with thousands of coroutines. We measured the CPU overhead to switch between coroutines to be 13–20 ns.

In FaSST, each thread creates one RPC endpoint that is shared by the coroutines spawned by the thread. One coroutine serves as the master; the remaining are workers. Worker coroutines only run application logic and issue RPC requests to remote machines, where they are processed by the master coroutine of the thread handling the request. The master coroutine polls the network to identify any newly-arrived request or response packets. The master computes and sends responses for request packets. It buffers response packets received for each worker until all needed responses are available, at which time it invokes the worker.

4.2 RPC interface and optimizations

A worker coroutine operates on batches of $b \geq 1$ requests, based on what the application logic allows. The worker begins by first creating new requests without performing network I/O. For each request, it specifies the request type (e.g., access a particular database table, transaction logging, etc.), and the ID of destination machine. After creating a batch of requests, the worker invokes an RPC function to send the request messages. Note that an RPC request specifies the destination machine, not the destination thread; FaSST chooses the destination thread as the local thread's ID–based peer on the destination machine. Restricting RPC communication to between thread peers improves FaSST's scalability by reducing the number of coroutines that can send requests to a thread (Section 4.4).

Request batching. Operating on batches of requests has several advantages. First, it reduces the number of NIC Doorbells the CPU must ring from b to 1, saving CPU cycles. Second, it allows the RPC layer to coalesce messages sent to the same destination machine. This is particularly useful for multi-key transactions that access multiple tables with same primary key, e.g., in the SmallBank benchmark (Section 6). Since our transaction layer partitions tables by a hash of the primary key, the table access requests are sent in the same packet. Third, batching reduces coroutine switching overhead: the master yields to a worker only after receiving responses for all b requests, reducing switching overhead by a factor of b.

Response batching: Similar to request batching, FaSST also uses batching for responses. When the master coroutine polls the NIC for new packets, it typically receives more than one packet. On receiving a batch of B request packets, it invokes the request handler for each request, and assembles a batch of B response packets. These responses are sent using one Doorbell. Note that the master does not wait for a batch of packets to accumulate before sending responses to avoid adding latency.

Cheap RECV posting: FaSST's RPCs use two-sided verbs, requiring RECVs to be posted on the RECV queue before an incoming SEND arrives. On our InfiniBand hardware, posting RECVs requires creating descriptors in the host-memory RECV queue, and updating the queue's host-memory head pointer. No CPU-initiated PCIe transactions are required as the NIC fetches the descriptors using DMA reads. In FaSST, we populate the RECV queue with descriptors once during initialization, after which the descriptors are not accessed by the CPU; new RECVs re-use descriptors in a circular fashion, and can be posted with a single write to the cached head pointer. Doing so required modifying the NIC's device driver, but it saves CPU cycles.

It is interesting to note that in FaSST, the NIC's RECV descriptor DMA reads are redundant, since the descriptors never change after initialization. Avoiding these DMA reads may be possible with device firmware changes or with future programmable NICs; doing so will likely give FaSST's RPCs a large throughput boost [16].

4.3 Detecting packet loss

The master coroutine at each thread detects packet loss for RPCs issued by its worker coroutines. The master tracks the progress of each worker by counting the number of responses received for the worker. A worker's progress counter stagnates if and only if one of the worker's RPC packets (either the request or the response) is lost: If a packet is lost, the master never receives all responses for the worker; it never invokes the worker again, preventing it from issuing new requests and receiving more responses. If no packet is lost, the master eventually receives all responses for the worker. The worker gets invoked and issues new requests—we do not allow workers to yield to the master without issuing RPC requests.

If the counter for a worker does not change for `timeout` seconds, the master assumes that the worker suffered a packet loss. On suspecting a loss, the master kills the FaSST process on its machine (Section 5.1). Note that, before it is detected, a packet loss affects only the progress of one worker, i.e., other workers can successfully commit transactions until the loss is detected. This allows us to use a large value for `timeout` without affecting FaSST's availability. We currently set `timeout` to one second. In our experiments with 50+ nodes, we did not observe a false positive with this timeout value. We observed false positives with significantly smaller timeout values such as 100 ms. This can happen, for instance, if the thread handling the RPC response gets preempted [12].

4.4 RPC limitations

Although FaSST aims to provide general-purpose RPCs, we do currently not support workloads that require messages larger than the network's MTU (4 KB on our InfiniBand network). These workloads are likely to be bottlenecked by network bandwidth with both RPC- and one-sided RDMA-based designs, achieving similar performance. This limitation can be addressed in several performance-neutral ways if needed.[2]

FaSST also restricts each coroutine to one message per destination machine per batch; the message, however, can contain multiple coalesced requests. This restriction is required to keep the RECV queues small so that they can be cached by the NIC. Consider a cluster with N nodes, T threads per node, and c coroutines per thread. For a given thread, there are N peer threads, and $N * c$ coroutines that can send requests to it. At any time, each thread must provision as many RECVs in its RECV queue as the number of requests that can be sent to it. Allowing each coroutine m messages per destination machine requires maintaining $(N*c*m)$ RECVs per RECV queue. A fairly large cluster with $N = 100$, $c = 20$, and $T = 14$ requires 14 RECV queues of size $2000*m$ at each machine. $m = 1$ was sufficient for our workloads and worked well in our experiments, but significantly larger values of m reduce RPC performance by causing NIC cache thrashing.

Supporting a larger cluster may require reducing RECV queue size. This can be achieved by reducing the number of requests allowed from a local thread to a particular remote machine from c to some smaller number; a coroutine yields if its thread's budget for a remote machine is temporarily exhausted. This will work well with large clusters and workloads without high skew, where the probability of multiple coroutines sending requests to the same remote machine is small.

4.5 Single-core RPC performance

We showed in Section 3.3 that FaSST RPCs provide good per-NIC throughput. We now show that they also provide good single-core throughput. To measure per-core throughput, we run one thread per machine, 20 coroutines per thread, and use 32-byte RPCs. We use all 11 available machines on CIB; we use 11 machines on CX3 for comparison. We evaluate RPC performance with multiple request batch sizes. To prevent request coalescing by our RPC layer, we choose a different machine for each request in the batch.

For our RPC baseline, we use a request batch size of one, and disable response batching. We then enable the request batching, cheap RECV posting, and response

[2]Infrequent small but > 4KB messages could be segmented in the RPC layer. Alternately, a three-message exchange wherein the sender requests that the recipient use a one-sided READ to obtain a large message payload could be used.

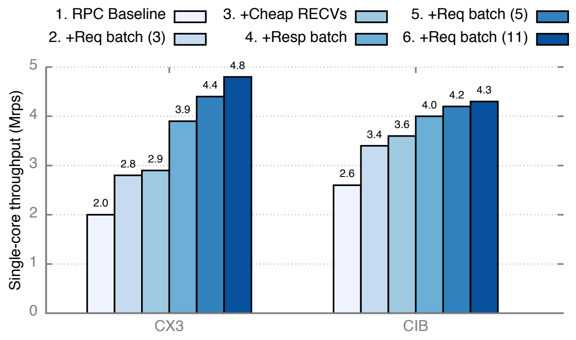


Figure 4: Per-core RPC throughput as optimizations 2–6 are added

batching optimizations in succession. Figure 4 shows the results from this experiment. Note that the batching optimizations do not apply to READs, because Doorbells cannot be batched across the connected QPs. For brevity, we discuss only CIB here.

Even without any optimizations, FaSST RPCs are more CPU-efficient than READs with QP sharing: our baseline achieves 2.6 Mrps, whereas READs achieve up to 2 Mrps with QP sharing between 3 or more threads (Figure 3). With a request batch size of 3 and all optimizations enabled, FaSST RPCs achieve 4 Mrps—2x higher than READs. Peak RPC throughput with one request per batch is 3.4 Mrps (not shown).

With 11 requests per batch, FaSST RPCs achieve 4.3 Mrps. At this request rate, each CPU core issues 17.2 million verbs per second on average: 4.3 million SENDs each for requests and responses, and 8.6 million for their RECVs. This large advantage over one-sided READs (which achieve 2 million verbs per second) arises from FaSST's use of datagram transport, which allows exclusive access to QPs and Doorbell batching.

Comparison with FaRM RPCs: FaRM's RPCs achieve up to 5 Mrps with one ConnectX-3 NIC and 16 CPU cores [11]. Their throughput does not increase noticeably when another ConnectX-3 NIC is added [12], so we expect them to provide $\approx$ 5 Mrps with a Connect-IB NIC. FaSST RPCs can achieve 34.9–40.9 Mrps (Figure 1), i.e., up to 8x higher throughput per NIC. FaRM's RPCs achieve $5/16 = 0.31$ Mrps per core; FaSST can achieve 3.4–4.3 Mrps per core depending on the request batch size (up to 13.9x higher).

5 Transactions

FaSST provides transactions with serializability and durability on partitioned distributed data stores. FaSST's data stores map 8-byte keys to opaque application-level objects. Each key is associated with an 8-byte header, consisting of a lock bit, and a 63-bit version number. The header is used for concurrency control and for ordering commit

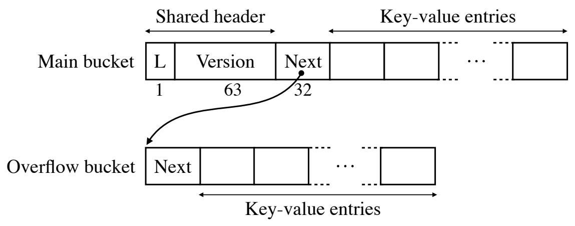


Figure 5: Layout of main and overflow buckets in our MICA-based hash table

log records during recovery. Several keys can map to the same header.

We have implemented transactions for an unordered key-value store based on MICA [18]. The key-value store uses a hash table composed of associative buckets (Figure 5) with multiple (7–15) slots to store key-value items. Each key maps to a *main* bucket. If the number of keys mapping to a main bucket exceeds the bucket capacity, the main bucket is dynamically linked to a chain of *overflow* buckets. The header for all keys stored in a main bucket and its linked overflow buckets is maintained in the main bucket.

In FaSST, worker coroutines run the transaction logic and act as transaction coordinators. FaSST's transaction protocol is inspired by FaRM's, with some modifications for simplicity. FaSST uses optimistic concurrency control and two-phase commit for distributed atomic commit, and primary-backup replication to support high availability. We use the Coordinator Log [24] variant of two-phase commit for its simplicity. Figure 6 summarizes FaSST's transaction protocol. We discuss the protocol's phases in detail below. All messages are sent using FaSST RPCs. We denote the set of keys read and written by the transaction by R (read set) and W (write set) respectively. We assume that the transaction first reads the keys it writes, i.e., $W \subseteq R$.

1. Read and lock: The transaction coordinator begins execution by reading the header and value of keys from their primaries. For a key in W, the coordinator also requests the primary to lock the key's header. The flexibility of RPCs allows us to read and lock keys in a single round trip. Achieving this with one-sided RDMA requires 2 round trips: one to lock the key using an ATOMIC operation, and one to read its value [7]. If any key in R or W is already locked, the coordinator aborts the transaction by sending unlock RPCs for successfully locked keys.

2. Validate: After locking the write set, the coordinator checks the versions of its read set by requesting the versions of R again. If any key is locked or its version has changed since the first phase, the coordinator aborts the transaction.

3. Log: If validation succeeds, the transaction can commit. To commit a transaction, the coordinator replicates

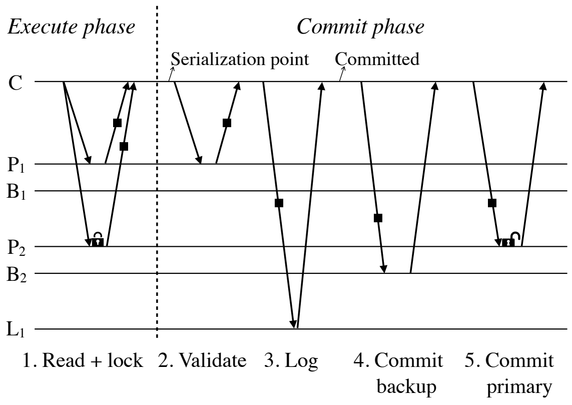


Figure 6: FaSST's transaction protocol with tolerance for one node failure. P_1 and P_2 are primaries and B_1 and B_2 are their backups. C is the transaction coordinator, whose log replica is L_1. The solid boxes denote messages containing application-level objects. The transaction reads one key from P_1 and P_2, and updates the key on P_2.

its commit log record at $f + 1$ log replicas so that the transaction's commit decision survives f failures. The coordinator's host machine is always a log replica, so we send f RPCs to remote log replicas. The commit log record contains W's key-value items and their fetched versions.

4. Commit backup: If logging succeeds, the coordinator sends update RPCs to backups of W. It waits for an ACK from each backup before sending updates to the primaries. This wait ensures that backups process updates for a bucket in the same order as the primary. This ordering is not required in FaRM, which can drop out-of-order bucket updates as each update contains the contents of the entire bucket. FaSST's updates contain only one key-value item and are therefore smaller, but cannot be dropped.

5. Commit primary: After receiving all backup ACKs, the coordinator sends update RPCs to the primaries of W. On receiving an update, a primary updates the key's value, increments its version, and unlocks the key.

Similar to existing systems [12, 28], FaSST omits validation and subsequent phases for single-key read-only transactions.

5.1 Handling failures and packet loss

Currently, the FaSST implementation provides serializability and durability, but not high availability. Similar to prior single-node transaction systems [27], we have implemented the normal case datapath (logging and replication) to the extent that fast recovery is possible, but we have not implemented the actual logic to recover from a machine failure. We assume that FaRM's mechanisms to detect and recover from machine failures, such as leases, cluster membership reconfiguration, log replay, and re-replication of lost data can be adapted to FaSST; we discuss how packet losses can be handled below. Note that our implementation is insensitive to packet reordering since each RPC message is smaller than the network's MTU.

We convert a packet loss to a machine failure by killing the FaSST process on the machine that detects a lost RPC packet (Section 4.3). The transaction affected by the lost packet will not make progress until the killed FaSST process is detected (e.g., via leases); then the transaction's commit/abort decision will be handled by the recovery mechanism. This basic scheme can be improved (e.g., the victim node can be re-used to avoid data re-replication since it need not reboot), but that is not the focus of our work.

In Section 3.4, we measured the packet loss rate of our network at less than one in 50 PB of data. Since we did not actually lose a packet, the real loss rate may be much lower, but we use this upper-bound rate for a ballpark availability calculation. In a 100-node cluster where each node is equipped with 2x56 Gbps InfiniBand and transfers data at full-duplex bandwidth, 50 PB will be transferred in approximately 5 hours. Therefore, packet losses will translate to less than 5 machine failures per day. Assuming that each failure causes 50 ms of downtime as in FaRM [12], FaSST will achieve five-nines of availability.

5.2 Implementation

We now discuss details of FaSST's transaction implementation. Currently, FaSST provides transactions on 8-byte keys and opaque objects up to 4060 bytes in size. The value size is limited by our network's MTU (4096 bytes) and the commit record header overhead (36 bytes). To extend a single-node data store for distributed transactions, a FaSST user writes RPC request handlers for pre-defined key-value requests (e.g., get, lock, put, and delete). This may require changes to the single-node data store, such as supporting version numbers. The user registers database tables and their respective handlers with the RPC layer by assigning each table a unique RPC request type; the RPC subsystem invokes a table's handler on receiving a request with its table type.

The data store must support concurrent local read and write access from all threads in a node. An alternate design is to create exclusive data store partitions per thread, instead of per-machine partitions as in FaSST. As shown in prior work [18], this alternate design is faster for local data store access since threads need not use local concurrency control (e.g., local locks) to access their exclusive partition. However, when used for distributed transactions, it requires the RPC subsystem to support all-to-all communication between threads, which reduces scalability by amplifying the required RECV queue size (Section 4.4). We chose to sacrifice higher CPU efficiency on small clusters for a more pressing need: cluster-level scalability.

	Nodes	**NICs**	**CPUs (cores used, GHz)**
FaSST (CX3)	50	1	1x E5-2450 (8, 2.1 GHz)
FaRM [12]	90	2	2x E5-2650 (16, 2.0 GHz)
DrTM+R [7]	6	1	1x E5-2450-v3 (8, 2.3 GHz)

Table 3: Comparison of clusters used to compare published numbers. The NIC count is the number of ConnectX-3 NICs. All CPUs are Intel® Xeon ® CPUs. DrTM+R's CPU has 10 cores but their experiments use only 8 cores.

5.2.1 Transaction API

The user writes application-level transaction logic in a worker coroutine using the following API.

`AddToReadSet(K, *V)` and **`AddToWriteSet(K, *V, mode)`** enqueue key `K` to be fetched for reading or writing, respectively. For write set keys, the write mode is either insert, update, or delete. After the coroutine returns from `Execute` (see below) the value for key `K` is available in the buffer `V`. At this point, the application's transaction logic can modify `V` for write set keys to the value it wishes to commit.

`Execute()` sends the execute phase-RPCs of the transaction protocol. Calling `Execute` suspends the worker coroutine until all responses are available. Note that the `AddToReadSet` and `AddToWriteSet` functions above do not generate network messages immediately: requests are buffered until `Execute` is called. This allows the RPC layer to send all requests in with one Doorbell, and coalesce requests sent to the same remote machine. Applications can call `Execute` multiple times in one transaction after adding more keys. This allows transactions to choose new keys based on previously fetched keys.

`Execute` fails if a read or write set key is locked. In this case, the transaction layer returns failure to the application, which then must call `Abort`.

`Commit()` runs the commit protocol, including validation, logging and replication, and returns the commit status. **`Abort()`** sends unlock messages for write set keys.

6 Evaluation

We evaluate FaSST using 3 benchmarks: an object store, a read-mostly OLTP benchmark, and a write-intensive OLTP benchmark. We use the simple object store benchmark to measure the effect of two factors that affect FaSST's performance: multi-key transactions and the write-intensiveness of the workload. All benchmarks include 3-way logging and replication, and use 14 threads per machine.

We use the other two benchmarks to compare against two recent RDMA-based transaction systems, FaRM [12] and DrTM+R [7]. Unfortunately, we are unable do a direct comparison by running these systems on our clusters. FaRM is not open-source, and DrTM+R depends on Intel's Restricted Transactional Memory (RTM). Intel's RTM is disabled by default on Haswell processors due to a hardware bug that can cause unpredictable behavior. It can be re-enabled by setting model-specific registers [28], but we were not permitted to do so on the CIB cluster.

For a comparison against published numbers, we use the CX3 cluster which has less powerful hardware (NIC and/or CPU) than used in FaRM and DrTM+R; Table 3 shows the differences in hardware. We believe that the large performance difference between FaSST and other systems (e.g., 1.87x higher than FaRM on TATP with half the hardware resources; 1.68x higher than DrTM+R on SmallBank without locality assumptions) offsets performance variations due to system and implementation details. We also use the CIB cluster in our evaluation to show that FaSST can scale up to more powerful hardware.

TATP is an OLTP benchmark that simulates a telecommunication provider's database. It consists of 4 database tables with small key-value pairs up to 48 bytes in size. TATP is read-intensive: 70% of TATP transactions read a single key, 10% of transactions read 1–4 keys, and the remaining 20% of transactions modify keys. TATP's read-intensiveness and small key-value size makes it well-suited to FaRM's design goal of exploiting remote CPU bypass: 80% of TATP transactions are read-only and do not involve the remote CPU. Although TATP tables can be partitioned intelligently to improve locality, we do not do so (similar to FaRM).

SmallBank is a simple OLTP benchmark that simulates bank account transactions. SmallBank is write-intensive: 85% of transactions update a key. Our implementation of SmallBank does not assume data locality. In DrTM+R, however, single-account transactions (comprising 60% of the workload) are initiated on the server hosting the key. Similarly, only a small fraction (< 10%) of transactions that access two accounts access accounts on different machines. These assumptions make the workload well-suited to DrTM+R's design goal of optimizing local transactions by using hardware transactional memory, but they save messages during transaction execution and commit. We do not make either of these assumptions and use randomly chosen accounts in all transactions.

Although the TPC-C benchmark [26] is a popular choice for evaluating transaction systems, we chose not to include it in our benchmarks for two reasons. First, TPC-C has a high degree of locality: only around 10% of transactions (1% of keys) access remote partitions. The speed of local transactions and data access, which our work does not focus on, has a large impact on TPC-C performance. Second, comparing performance across TPC-C implementations is difficult. This is due to differences in data structures (e.g., using hash tables instead of B-Trees

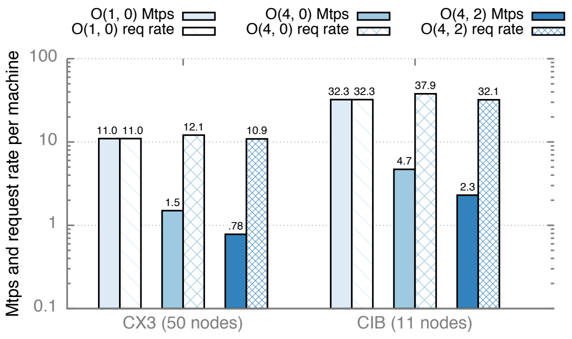


Figure 7: Object store performance. The solid and patterned bars show transaction throughput and RPC request rate, respectively. The Y axis is in log scale.

for some tables), interaction of the benchmark with system optimizations (e.g., FaRM and DrTM+R use caching to reduce READs, but do not specify cache hit rates), and contention level (DrTM+R uses 1 TPC-C "warehouse" per thread whereas FaRM uses ≈ 7, which may reduce contention).

6.1 Object store

We create an object store with small objects with 8-byte keys and 40-byte values. We scale the database by using 1 million keys per thread in the cluster. We use workloads with different read and write set sizes to evaluate different aspects of FaSST. An object store workload in which transactions read r keys, and update w of these keys (on average) is denoted by $O(r,w)$; we use $O(1,0)$, $O(4,0)$, and $O(4,2)$ to evaluate single-key read-only transactions, multi-key read-only transactions, and multi-key read-write transactions. All workloads choose keys uniformly at random; to avoid RPC-level coalescing, keys are chosen such that their primaries are on different machines. Figure 7 shows FaSST's performance on the object store workloads on the two clusters.

6.1.1 Single-key read-only transactions

With $O(1,0)$ FaSST achieves 11.0 million transactions per second (Mtps) per machine on CX3. FaSST is bottlenecked by the ConnectX-3 NIC: this throughput corresponds to 11.0 million RPC requests per second (Mrps), which is 96.5% of the NIC's maximum RPC throughput in this scenario.

On CIB, FaSST achieves 32.3 Mtps/machine and is CPU-limited. This is because $O(1,0)$ does not allow Doorbell batching for requests, leading to low per-core throughput. Although CIB's CPUs can saturate the NIC without request Doorbell batching for an RPC microbenchmark that requires little computation (Section 3.3.1), they cannot do so for $O(1,0)$ which requires key-value store accesses.

Comparison: FaRM [12] reports performance for the $O(1,0)$ workload. FaRM uses larger, 16-byte keys and

32-byte values. Our FaSST implementation currently supports only 8-byte keys, but we use larger, 40-byte values to keep the key-value item size identical. Using 16-byte keys is unlikely to change our results.[3]

FaRM achieves 8.77 Mtps/machine on a 90-node cluster with $O(1,0)$. It does not saturate its 2 ConnectX-3 NICs and is instead bottlenecked by its 16 CPU cores. FaSST achieves 1.25x higher per-machine throughput with 50 nodes on CX3, which has close to half of FaRM's hardware resources per node (Table 3). *Although $O(1,0)$ is well-suited to FaRM's design goal of remote CPU bypass (i.e., no transaction involves the remote CPU), FaRM performs worse than FaSST.* Note that with FaRM's hardware—2 ConnectX-3 NICs and 16 cores—FaSST will deliver higher performance; based on our CIB results, we expect FaSST to saturate the two ConnectX-3 NICs and outperform FaRM by 2.5x.

6.1.2 Multi-key transactions

With multi-key transactions, FaSST reduces per-message CPU use by using Doorbell batching for requests. With $O(4,0)$, FaSST achieves 1.5 and 4.7 Mtps/machine on CX3 and CIB, respectively. (The decrease in Mtps from $O(1,0)$ is because the transactions are larger.) Similar to $O(1,0)$, FaSST is NIC-limited on CX3. On CIB, however, although FaSST is CPU-limited with $O(1,0)$, it becomes NIC-limited with $O(4,0)$. With $O(4,0)$ on CIB, each machine generates 37.9 Mrps on average, which matches the peak RPC throughput achievable with a request batch size of 4.

With multi-key read-write transactions in $O(4,2)$, FaSST achieves 0.78 and 2.3 Mtps/machine on CX3 and CIB, respectively. FaSST is NIC-limited on CX3. On CIB, the bottleneck shifts to CPU again because key-value store inserts into the replicas' data stores are slower than lookups.

Comparison: FaRM does not report object store results for multi-key transactions. However, as FaRM's connected transport does not benefit from Doorbell batching, we expect the gap between FaSST's and FaRM's performance to increase. For example, while FaSST's RPC request rate increases from 32.3 Mrps with $O(1,0)$ to 37.9 Mrps with $O(4,0)$, the change in FaRM's READ rate is likely to be negligible.

6.2 TATP

We scale the TATP database size by using one million TATP "subscribers" per machine in the cluster. We use all CPU cores on each cluster and increase the number of machines to measure the effect of scaling. Figure 8 shows the throughput on our clusters. On CX3, FaSST achieves 3.6 Mtps/machine with 3 nodes (the minimum required for 3-way replication), and 3.55 Mtps/machine with 50 nodes. On CIB, FaSST's throughput increases to 8.7 Mtps/machine with 3–11 nodes. In both cases, FaSST's throughput scales linearly with cluster size.

[3]On a single node, FaSST's data store (MICA) delivers similar GET throughput (within 3%) for these two key-value size configurations. Throughput is *higher* with 16-byte keys, which could be because MICA's hash function uses fewer cycles.

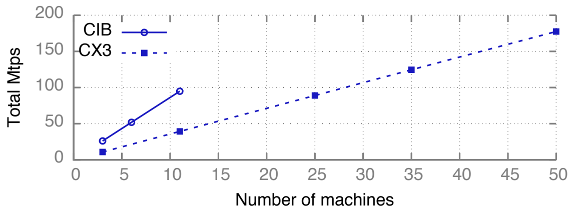


Figure 8: TATP throughput

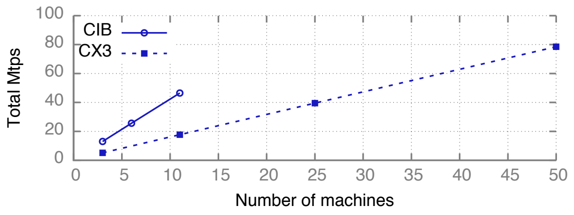


Figure 9: SmallBank throughput

Comparison: FaRM [12] reports 1.55 Mtps/machine for TATP on a 90-node cluster. With a smaller 50-node cluster, however, FaRM achieves higher throughput (≈ 1.9 Mtps/machine) [1]. On 50 nodes on CX3, FaSST's throughput is 87% higher. Compared to $O(1,0)$, the TATP performance difference between FaSST and FaRM is higher. TATP's write transactions require using FaRM's RPCs, which deliver 4x lower throughput than FaRM's one-sided READs, and up to 8x lower throughput than FaSST's RPCs (Section 4.5).

6.3 SmallBank

To scale the SmallBank database, we use 100,000 bank accounts per thread. 4% of the total accounts are accessed by 90% of transactions. (Despite the skew, the workload does not have significant contention due to the large number of threads, and therefore "bank accounts" in the workload/cluster.) This configuration is the same as in DrTM [28]. Figure 9 shows FaSST's performance on our clusters. FaSST achieves 1.57–1.71 Mtps/machine on CX3, and 4.2–4.3 Mtps/machine on CIB, and scales linearly with cluster size.

Comparison: DrTM+R [7] achieves 0.93 Mtps/machine on a cluster similar to CX3 (Table 3), but with more powerful CPUs. FaSST outperforms it by over 1.68x on CX3, and over 4.5x on CIB. DrTM+R's lower performance comes from three factors. First, ATOMICs lead to a fundamentally slower protocol. For example, excluding logging and replication, for a write-set key, DrTM+R uses four separate messages to read, lock, update, and unlock the key; FaSST uses only two messages. Second, as shown

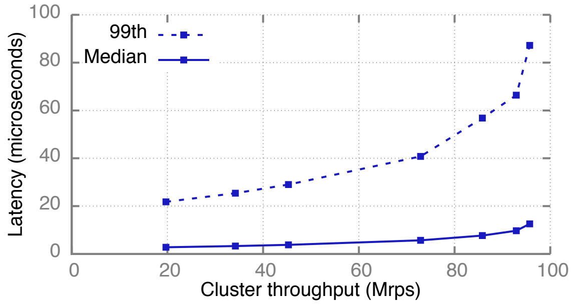


Figure 10: TATP latency on CIB

in our prior work, ATOMICs perform poorly (up to 10x worse than READs) on the ConnectX-3 NICs [16] used in DrTM+R; evaluation on Connect-IB NICs may yield better performance, but is unlikely to outperform FaSST because of the more expensive protocol. Third, DrTM+R does not use queue pair sharing, so their reported performance may be affected by NIC cache misses.

6.4 Latency

For brevity, we discuss FaSST's latency only for TATP on CIB. Figure 10 shows FaSST's median and 99^{th} percentile latency for successfully committed TATP transactions. To plot a throughput-latency curve, we vary the request load by increasing the number of worker coroutines per thread from 1 to 19; each machine runs 14 threads throughout. We use at most 19 worker coroutines per thread to limit the RECV queue size required on a (hypothetical) 100-node cluster to 2048 RECVs (Section 4.4). Using the next available RECV queue size with 4096 RECVs can cause NIC cache misses for some workloads. With one worker coroutine per thread, the total transaction throughput is 19.7 Mtps with 2.8 µs median latency and 21.8 µs 99^{th} percentile latency. Since over 50% of committed transactions in TATP are single-key reads, FaSST's median latency at low load is close to the network's RTT. This shows that our batching optimizations do not add noticeable latency. With 19 worker coroutines per thread, cluster throughput increases to 95.7 Mtps, and median and 99^{th} percentile latency increase to 12.6 µs and 87.2 µs, respectively.

7 Future trends

7.1 Scalable one-sided RDMA

A key limitation of one-sided RDMA on current commodity hardware is its low scalability. This limitation itself comes from the fundamentally connection-oriented nature of the Virtual Interface Architecture. Two attempts at providing scalable one-sided RDMA are worth mentioning.

DCT: Mellanox's Dynamically Connected Transport [2] (DCT) preserves their core connection-oriented design, but dynamically creates and destroys one-to-one connections. This provides software the illusion of using one QP to communicate with multiple remote machines, but at a prohibitively large performance cost for our workloads: DCT requires three additional network messages when the target machine of a DCT queue pair changes: a disconnect packet to the current machine, and a two-way handshake with the next machine to establish a connection [8]. In a high fanout workload such as distributed OLTP, this increases the number of packets associated with each RDMA request by around 1.5x, reducing performance.

A detailed evaluation of DCT on CIB is available in FaSST's source code repository. Here, we discuss DCT's performance in the READ rate benchmark used in Section 3.3.1. We use 6 machines and 14 threads per machine, which issue 32-byte READs to machines chosen uniformly at random. We vary the number of outstanding READs per thread, and the number of DCT QPs used by each thread. (Using only one DCT QP per thread limits its throughput to approximately one operation per multiple RTTs, since a QP cannot be used to READ from multiple machines concurrently. Using too many DCT QPs causes cache NIC misses.) We achieve only up to 22.9 Mrps per machine—55.3% lower than the 51.2 Mrps achievable with standard READs over the RC transport (Figure 1).

Portals: Portals is a less-widespread RDMA specification that provides scalable one-sided RDMA using a connectionless design [4]. Recently, a hardware implementation of Portals—the Bull eXascale Interconnect [10]—has emerged, currently available only to HPC customers. The availability of scalable one-sided RDMA may reduce the performance gap between FaSST and FaRM/DrTM+R. However, even with scalable one-sided RDMA, transaction systems that use it will require large or multiple READs to access data stores, reducing performance. Further, it is likely that RPC implementations that use scalable one-sided WRITEs will provide even better performance than FaSST's two-sided RPCs by avoiding the NIC and PCIe overhead of RECVs. However, WRITE-based RPCs do not scale well on current commodity hardware.

In this paper, we took an extreme position where we used only RPCs, demonstrating that remote CPU bypass is not required for high performance transactions, and that a design using optimized RPCs can provide better performance. *If scalable one-sided RDMA becomes commonly available in the future, the best design will likely be hybrid of RPCs and remote bypass, with RPCs used for accessing data structures during transaction execution, and scalable one-sided WRITEs used for logging and replication during transaction commit.*

7.2 More queue pairs

Our experiments show that the newer Connect-IB NIC can cache a larger number of QPs than ConnectX-3 (Figure 2). Just as advances in technology yield NICs that are faster and have more/better cache, newer CPUs will also have more cores. We showed earlier that sharing QPs between only 2 threads causes CPU efficiency to drop by several factors (Figure 3). Avoiding QP sharing with next-generation CPUs (e.g., 28 cores in Intel's upcoming Skylake processors) on a 100-node cluster will require NICs that can cache 2800 QPs—7 times more than Connect-IB's 400 QPs. This trend lends additional support to datagram-based designs.

7.3 Advanced one-sided RDMA

Future NICs may provide advanced one-sided RDMA operations such as multi-address atomic operations, and B-Tree traversals [23]. Both of these operations require multiple PCIe round trips, and will face similar flexibility and performance problems as one-sided RDMA (but over the PCIe bus) if used for high-performance distributed transactions. On the other hand, we believe that "CPU onload" networks such as Intel's 100 Gbps OmniPath [6] are well-suited for transactions. These networks provide fast messaging over a reliable link layer, but not one-sided RDMA, and are therefore cheaper than "NIC offload" networks such as Mellanox's InfiniBand. FaSST requires only messaging, so we expect our design to work well over OmniPath.

8 Related work

High-performance RDMA systems: FaSST draws upon our prior work on understanding RDMA performance [16], where we demonstrated the effectiveness of Doorbell batching for RDMA verbs. A number of recent systems have used one-sided verbs to build key-value stores and distributed shared memory [20, 21, 11, 21]. These systems demonstrate that RDMA is now a practical primitive for building non-HPC systems, though their one-sided designs introduce additional complexities and performance bottlenecks. FaSST's two-sided RPC approach generalizes our approach in HERD [15]. HERD used a hybrid of unreliable one-sided and two-sided RDMA to implement fast RPCs in a client-server setting; FaSST extends this model to a symmetric setting and further describes a technique to implement reliability.

Distributed transactions in the datacenter: Like FaSST, FaRM [12] uses primary-backup replication and optimistic concurrency control for transactions. FaRM's design (unlike FaSST) is specialized to work with their desire to use one-sided RDMA verbs. FaRM also provides fast failure detection and recovery, and a sophisticated programming model, which was not a goal of this work. Several projects use one-sided ATOMICs for transactions [28, 7, 5]. Though an attractive primitive, ATOMICs are slow on current NICs (e.g., ConnectX-3 serializes all ATOMIC operations [16]), use connected QPs, and fundamentally require more messages than an RPC-based approach (e.g., separate messages are needed to read and lock a key). Calvin [25] uses conventional networking without kernel bypass, and is designed around avoiding distributed commit. Designs that use fast networks, however, can use traditional distributed commit protocols to achieve high performance [12, 28].

9 Conclusion

FaSST is a high-performance, scalable, distributed in-memory transaction processing system that provides serializability and durability. FaSST achieves its performance using FaSST RPCs, a new RPC design tailored to the properties of modern RDMA hardware that uses two-sided verbs and datagram transport. It rejects one of the seemingly most attractive properties of RDMA—CPU bypass—to keep its communication overhead low and its system design simple and fast. The combination allows FaSST to outperform recent RDMA-based transactional systems by 1.68x–1.87x with fewer resources and making fewer workload assumptions. Finally, we provide the first large-scale study of InfiniBand network reliability, demonstrating the rarity of packet loss on such networks.

Acknowledgments We are tremendously grateful to Joseph Moore and NetApp for providing continued access to the CIB cluster. We thank Miguel Castro for initial discussions that shaped this project, and for his feedback on the paper. We also thank Milo Martin for his feedback that helped improve the paper, Chao Xin for conducting initial performance measurements, Hyeontaek Lim for his generous support with running MICA, and Y. Charlie Hu for shepherding. Emulab [29] and PRObE [14] resources were used in our experiments. PRObE is supported in part by NSF awards CNS-1042537 and CNS-1042543 (PRObE). This work was supported by funding from the National Science Foundation under awards 1345305, 1314721, and 1535821, and by Intel via the Intel Science and Technology Center for Cloud Computing (ISTC-CC).

References

[1] Private communication with FaRM's authors.

[2] Mellanox Connect-IB product brief. http://www.mellanox.com/related-docs/prod_adapter_cards/PB_Connect-IB.pdf, 2015.

[3] Mellanox OFED for Linux user manual. http://www.mellanox.com/related-docs/prod_software/Mellanox_OFED_Linux_User_Manual_v2.2-1.0.1.pdf, 2015.

[4] B. W. Barrett, R. Brightwell, S. Hemmert, K. Pedretti, K. Wheeler, K. Underwood, R. Riesen, A. B. Maccabe, and T. Hudson. The Portals 4.0 network programming interface november 14, 2012 draft.

[5] C. Binnig, A. Crotty, A. Galakatos, T. Kraska, and E. Zamanian. The end of slow networks: It's time for a redesign. In *Proc. VLDB*, New Delhi, India, Aug. 2016.

[6] M. S. Birrittella, M. Debbage, R. Huggahalli, J. Kunz, T. Lovett, T. Rimmer, K. D. Underwood, and R. C. Zak. Intel Omni-path architecture: Enabling scalable, high performance fabrics. In *Proceedings of the 2015 IEEE 23rd Annual Symposium on High-Performance Interconnects*, 2015.

[7] Y. Chen, X. Wei, J. Shi, R. Chen, and H. Chen. Fast and general distributed transactions using RDMA and HTM. In *Proc. 11th ACM European Conference on Computer Systems (EuroSys)*, Apr. 2016.

[8] D. Crupnicoff, M. Kagan, A. Shahar, N. Bloch, and H. Chapman. Dynamically-connected transport service, May 19 2011. URL https://www.google.com/patents/US20110116512. US Patent App. 12/621,523.

[9] 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 *Proc. 21st ACM Symposium on Operating Systems Principles (SOSP)*, Stevenson, WA, Oct. 2007.

[10] S. Derradji, T. Palfer-Sollier, J.-P. Panziera, A. Poudes, and F. W. Atos. The BXI interconnect architecture. In *Proceedings of the 2015 IEEE 23rd Annual Symposium on High-Performance Interconnects*, 2015.

[11] A. Dragojević, D. Narayanan, O. Hodson, and M. Castro. FaRM: Fast remote memory. In *Proc. 11th USENIX NSDI*, Seattle, WA, Apr. 2014.

[12] A. Dragojević, D. Narayanan, E. B. Nightingale, M. Renzelmann, A. Shamis, A. Badam, and M. Castro. No compromises: Distributed transactions with consistency, availability, and performance. In *Proc. 25th ACM Symposium on Operating Systems Principles (SOSP)*, Monterey, CA, Oct. 2015.

[13] D. Dunning, G. Regnier, G. McAlpine, D. Cameron, B. Shubert, F. Berry, A. M. Merritt, E. Gronke, and C. Dodd. The virtual interface architecture. *IEEE Micro*, pages 66–76, 1998.

[14] G. Gibson, G. Grider, A. Jacobson, and W. Lloyd. PRObE: A Thousand-Node Experimental Cluster for Computer Systems Research.

[15] A. Kalia, M. Kaminsky, and D. G. Andersen. Using RDMA efficiently for key-value services. In *Proc. ACM SIGCOMM*, Chicago, IL, Aug. 2014.

[16] A. Kalia, M. Kaminsky, and D. G. Andersen. Design guidelines for high-performance RDMA systems. In *Proc. USENIX Annual Technical Conference*, Denver, CO, June 2016.

[17] L. Lamport, D. Malkhi, and L. Zhou. Vertical Paxos and primary-backup replication. Technical report, Microsoft Research, 2009.

[18] H. Lim, D. Han, D. G. Andersen, and M. Kaminsky. MICA: A holistic approach to fast in-memory key-value storage. In *Proc. 11th USENIX NSDI*, Seattle, WA, Apr. 2014.

[19] Y. Mao, E. Kohler, and R. T. Morris. Cache craftiness for fast multicore key-value storage. In *Proc. 7th ACM European Conference on Computer Systems (EuroSys)*, Bern, Switzerland, Apr. 2012.

[20] C. Mitchell, Y. Geng, and J. Li. Using one-sided RDMA reads to build a fast, CPU-efficient key-value store. In *Proc. USENIX Annual Technical Conference*, San Jose, CA, June 2013.

[21] C. Mitchell, K. Montgomery, L. Nelson, S. Sen, and J. Li. Balancing CPU and network in the cell distributed B-Tree store. In *Proc. USENIX Annual Technical Conference*, Denver, CO, June 2016.

[22] J. Nelson, B. Holt, B. Myers, P. Briggs, L. Ceze, S. Kahan, and M. Oskin. Latency-tolerant software distributed shared memory. In *Proc. USENIX Annual Technical Conference*, Santa Clara, CA, June 2015.

[23] S. Raikin, L. Liss, A. Shachar, N. Bloch, and M. Kagan. Remote transactional memory, 2015. US Patent App. 20150269116.

[24] J. W. Stamos and F. Cristian. Coordinator log transaction execution protocol. *Distrib. Parallel Databases*, 1(4):383–408, Oct. 1993. ISSN 0926-8782. doi: 10.1007/BF01264014. URL http://dx.doi.org/10.1007/BF01264014.

[25] A. Thomson, T. Diamond, S.-C. Weng, K. Ren, P. Shao, and D. J. Abadi. Calvin: Fast distributed transactions for partitioned database systems. In *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data*, May

2012.

[26] TPC-C. TPC benchmark C. http://www.tpc.org/tpcc/, 2010.

[27] S. Tu, W. Zheng, E. Kohler, B. Liskov, and S. Madden. Speedy transactions in multicore in-memory databases. In *Proc. 24th ACM Symposium on Operating Systems Principles (SOSP)*, Farmington, PA, Nov. 2013.

[28] X. Wei, J. Shi, Y. Chen, R. Chen, and H. Chen. Fast in-memory transaction processing using RDMA and HTM. In *Proc. 25th ACM Symposium on Operating Systems Principles (SOSP)*, Monterey, CA, Oct. 2015.

[29] 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 *Proc. 5th USENIX OSDI*, pages 255–270, Boston, MA, Dec. 2002.

[30] H. Zhang, D. G. Andersen, A. Pavlo, M. Kaminsky, L. Ma, and R. Shen. Reducing the storage overhead of main-memory OLTP databases with hybrid indexes. In *Proc. ACM SIGMOD*, San Francisco, USA, June 2016.

NetBricks: Taking the V out of NFV

Aurojit Panda[†] Sangjin Han[†] Keon Jang[‡] Melvin Walls[†] Sylvia Ratnasamy[†] Scott Shenker[†⋆]
[†] UC Berkeley [‡] Google [⋆] ICSI

Abstract

The move from hardware middleboxes to software network functions, as advocated by NFV, has proven more challenging than expected. Developing new NFs remains a tedious process, requiring that developers repeatedly rediscover and reapply the same set of optimizations, while current techniques for providing isolation between NFs (using VMs or containers) incur high performance overheads. In this paper we describe NetBricks, a new NFV framework that tackles both these problems. For building NFs we take inspiration from modern data analytics frameworks (*e.g.,* Spark and Dryad) and build a small set of customizable network processing elements. We also embrace type checking and safe runtimes to provide isolation in software, rather than rely on hardware isolation. NetBricks provides the same memory isolation as containers and VMs, without incurring the same performance penalties. To improve I/O efficiency, we introduce a novel technique called zero-copy software isolation.

1 Introduction

Networks today are responsible for more than just forwarding packets, this additional functionality is implemented using "middleboxes". Middleboxes implement a wide range of functionality, including security (*e.g.,* firewalls, IDS/IPSs), performance (*e.g.,* caches, WAN optimizers) and support for new applications and protocols (*e.g.,* TLS proxies). Middlebox functionality was initially provided by dedicated hardware devices, and is in wide deployment today. A 2012 survey [44] found that in many networks there are equal numbers of middleboxes, switches and routers.

Approximately four years ago, many large carriers initiated an effort, called Network Function Virtualization (NFV), to replace hardware middleboxes with software implementations running in VMs [10]. This approach enabled middlebox functionality (called *Network Functions* or NFs) to be run on commodity servers and was supposed to bring several advantages including: (a) simplifying deployment, since deploying new functionality merely requires software changes; (b) simpler management using standard tools for managing VMs; (c) faster development, which now requires writing software that runs on commodity hardware; and (d) reduced costs by consolidating several NFs on a single machine. However, despite these promised advances, there has been little progress towards large-scale NF deployments. Our discussions with three major carriers revealed that they are only just beginning small scale test deployments (with 10-100s of customers) using simple NFs *e.g.,* firewalls and NATs.

The move from hardware middleboxes to software NFs was supposed to speed innovation, so why has progress been so slow? We believe this delay is because traditional approaches for both *building* and *running* NFs are a poor match for carrier networks, which have the following requirements: *performance*, NF deployments should be able to provide per-packet latencies on the order of 10s of μs, and throughput on the order of 10s of Gbps; *efficiency*, it should be possible to consolidate several NFs on a single machine; support for *chaining*, since each packet is typically processed by a sequence of NFs; the flexibility to run NFs manufactured by *multiple vendors*; and the ability to process packets from *multiple tenants* while providing some degree of isolation between them. Note that because many carriers provide middlebox services to their customers, the NFs supported by carriers include those that are commonly found in enterprise environments (*e.g.,* firewalls, NATs, IPS/IDSs, WAN optimizers, etc.) in addition to ones specific to carriers (*e.g.,* EPC, carrier-grade NAT).

Why do current tools for building and running NFs fall short of these requirements? In terms of *building* NFs, tools need to support both rapid-development (achieved through the use of high-level abstractions) and high performance (often requiring low-level optimizations). In other application domains, programming frameworks and models have been developed to allow developers to use high-level abstractions while the framework optimizes the implementations of those abstractions (ensuring high performance); the rise of data analytic frameworks (*e.g.,* Hadoop, Spark) is an example of this phenomenon. However, the state-of-the-art for NFV is much more primitive. There are programming models such as Click [27] that do not provide easily customizable low-level optimizations, and libraries such as

DPDK [23] that only provide highly-optimized packet I/O and low-level processing primitives (which alone is not sufficient to implement real-world NFs), but no approach that provides both high performance and rapid development. The result is that today NFV developers typically spend much time optimizing their code, which greatly slows development time and increases the likelihood for bugs.

The current approaches for *running* NFs is also inadequate. Isolation between NF is critical: memory isolation is essential for ensuring safety between NFs (which might come from different vendors); and performance isolation is essential to allow such deployments to serve multiple customers. Currently, NFV deployments rely on VMs and containers to provide isolation, but as we show in §5, they incur substantial performance overheads for simple NFs.

To address these inadequacies in the current NFV paradigm, we propose a very different approach for building and running NFs.[1] Our approach, called NetBricks, is clean-slate in that it requires rewriting NFs, but we do not see this as a significant drawback given the relative lack of progress towards NFV deployments. Our approach targets deployments in large carrier networks, but applies to other environments as well.

NetBricks provides both a programming model (for building NFs) and an execution environment (for running NFs). The programming model is built around a core set of high-level but customizable abstractions for common packet processing tasks; to demonstrate the generality of these abstractions and the efficiency of their implementations, we reimplemented 5 existing NFs in NetBricks and show that they perform well compared to their native versions. Our execution environment relies on the use of safe languages and runtimes for memory and fault isolation (similar to existing systems we rely on scheduling for performance isolation). Inter-process communication is also important in NF deployments, and IPC in these deployments must ensure that messages cannot be modified by an NF after being sent, a property we refer to as packet isolation. Current systems copy packets to ensure packet isolation, we instead use static check to provide this property without copies. The resulting design, which we call Zero-Copy Software Isolation (ZCSI), is the first to achieve memory and packet isolation with *no* performance penalty (in stark contrast to virtualization).

NetBricks is open source and is available at `https://netbricks.io`.

[1]Note that in addition to building and running NFs, one also has to *manage* them. There are separate and active efforts on this topic (discussed in §6) in both research [12, 37] and industry [11, 30] that are orthogonal to our concerns here.

2 Background and Motivation

In this section we provide a few more details on the problems with today's approaches to NFV, and then give a high-level description of how NetBricks resolves these problems. As in much of this paper, we separate the task of building NFs from the task of running them.

2.1 Building NFs

The vast majority of commercial NFs today make use of a fast I/O library (DPDK, netmap, etc.). While this greatly improves I/O performance, developers are responsible for all other code optimizations. The Click modular router (which can also make use of such libraries) enables developers to construct an NF by connecting together various packet processing modules (called elements). While Click does not limit how packets flow between elements, modules typically support only limited amount of customization through setting various parameters. Thus, when implementing new NF functionality, developers commonly need to implement new modules, and optimizing the performance of such a module is difficult and time-consuming.

Our approach differs in two respects. First, we limit the set of such modules to core functions such as packet parsing, processing payloads, bytestream processing, and the like. That is, rather than have developers deal with a large set of modules – trying to determine which best suit their needs in terms of optimization and generality – NetBricks focuses on a core set with well-known semantics and highly-optimized implementations.

Second, in order to provide the necessary generality, we allow these core modules to be customized through the use of User-Defined Functions (UDFs). This gives these modules great flexibility, while allowing NetBricks to use optimized implementations of these modules. We think this approach represents a sweet-spot in the tradeoff between flexibility and performance; yes, one can imagine NFs that would not be easily supported by the set of modules NetBricks provides, but all the common NFs we know of fit comfortably within NetBricks' range. NetBricks thus gives developers the flexibility they need, and operators the performance they want.

One can think of the relationship between Click and NetBricks to be analogous to the difference between MPI and Map Reduce. Both Click and MPI give developers a totally general framework in which to build their applications, but the developer must take on the task of optimizing the resulting code (unless they can reuse existing modules without change). In contrast, NetBricks and Map Reduce support only a more limited set of abstractions whose actions can be customized through user code.

2.2 Running NFs

Current NFV deployments typically involve NFs running in containers or VMs, which are then connected via vSwitch. In this setup VMs and containers provide isolation, ensuring that one NF cannot access memory belonging to another and the failure of an NF does not bring down another. The vSwitch abstracts NICs, so that multiple NFs can independently access the network, and is also responsible for transferring packets between NFs. This allows several NFs to be consolidated on a single physical machine and allows operators to "chain" several NFs together *i.e.,* ensure packets output from one NF are sent to another.

However these mechanisms carry a significant performance penalty. When compared to a single process with access to a dedicated NIC, per-core throughput drops by up to 3× when processing 64B (minimum size) packets using containers, and by up to 7× when using VMs. This gap widens when NFs are chained together; containers are up to 7× slower than a case where all NFs run in the same process, and VMs are up to 11× slower. Finally, running chaining multiple NFs in a single process is up to 6× faster than a case where each NF runs in a container (or VM) and is allocated its own core – this shows that adding cores does not address this performance gap. We provide more details on these results in §5.3.

The primary reason for this performance difference it that during network I/O packets must cross a hardware memory isolation boundary. This entails a context switch (or syscall), or requires that packets must cross core boundaries; both of which incur significant overheads. We avoid these overheads by relying on compile-time and runtime checks to enforce memory isolation in software. This is similar what was proposed by Singularity [20]. To further reduce packet I/O costs we use unique types [13] to implement safe 0-copy packet I/O between NFs. We call this technique Zero-Copy Software Isolation (ZCSI) and show that it provides low-overhead isolation for NFs.

3 Design

In this section we describe the design of NetBricks, starting with the programming abstractions and ending with the execution environment. We focus on NetBricks's architecture in this section, and present implementation notes in the next section.

3.1 Programming Abstractions

Network functions in NetBricks are built around several basic abstractions, whose behavior is dictated by user supplied functions (UDFs). An NF is specified as a directed graph with these abstractions as nodes. These abstractions fall into five basic categories – packet processing, bytestream processing, control flow, state management, and event scheduling – which we now discuss in turn.

Abstractions for Packet Processing: Each packet in NetBricks is represented by a structure containing (i) a stack of headers; (ii) the payload; and (iii) a reference to any per-packet metadata. Headers in NetBricks are structures which include a function for computing the length of the header based on its contents. Per-packet metadata is computed (and allocated) by UDFs and is used to pass information between nodes in an NF. UDFs operating on a packet are provided with the packet structure, and can access the last parsed header, along with the payload and any associated metadata. Each packet's header stack initially contains a "null" header that occupies 0 bytes.

We provide the following packet processing operators:

- **Parse:** Takes as input a header type and a packet structure (as described above). The abstraction parses the payload using the header type and pushes the resulting header onto the header stack and removes bytes representing the header from the payload.
- **Deparse:** Pops the bottom most header from the packet's header stack and returns it to the payload.
- **Transform:** This allows the header and/or payload to be modified as specified by a UDF. The UDF can make arbitrary changes to the packet header and payload, change packet size (adding or removing bytes from the payload) and can change the metadata or associate new metadata with the packet.
- **Filter:** This allows packet's meeting some criterion to be dropped. UDFs supplied to the filter abstraction return true or false. Filter nodes drops all packets for which the UDF returns false.

Abstractions for Bytestream Processing: UDFs operating on bytestreams are given a byte array and a flow structure (indicating the connection). We provide two operators for bytestream processing:

- **Window:** This abstraction takes four input parameters: window size, sliding increment, timeout and a stream UDF. The abstraction is responsible for receiving, reordering and buffering packets to reconstruct a TCP stream. The UDF is called whenever enough data has been received to form a window of the appropriate size. When a connection is closed or the supplied timeout expires, the UDF is called with all available bytes. By default, the Window abstraction also forwards all received packets (unmodified), allowing windows to be processed outside of the regular datapath. Alternatively, the operator can drop all received packets, and generate and send a modified output stream using the packetize node.
- **Packetize:** This abstraction allows users to convert

byte arrays into packets. Given a byte array and a header stack, the implementation segments the data into packets with the appropriate header(s) attached.

Our current implementations of these operators assume the use of TCP (*i.e.,* we use the TCP sequence numbers to do reordering, use FIN packets to detect a connection closing, and the *Packetize* abstraction applies headers by updating the appropriate TCP header fields), but we plan to generalize this to other protocols in the future.

Abstractions for Control Flow Control flow abstractions in NetBricks are necessary for branching (and merging branches) in the NF graph. Branching is required to implement conditionals (*e.g.,* splitting packets according to the destination port, etc.), and for scaling packet processing across cores. To efficiently scale across multiple cores, NFs need to minimize cross-core access to data (to avoid costs due to cache effects and synchronization); however, how traffic should be partitioned to meet this objective depends on the NF in question. Branching constructs in NetBricks therefore provide NF authors a mechanism to specify an appropriate partitioning scheme (*e.g.,* by port or destination address or connection or as specified by a UDF) that can be used by NetBricks's runtime. Furthermore, branching is often also necessary when chaining NFs together. Operators can use NetBricks's control flow abstractions to express such chaining behavior by dictating which NF a packet should be directed to next. To accomplish these various goals, NetBricks offers three control flow operators:

- **Group By:** Group By is used either to explicitly branch control flow within an NF or express branches in how multiple NFs are chained together. The group by abstraction takes as input the number of groups into which packets are split and a packet-based UDF which given a packet returns the ID of the group to which it belongs. NetBricks also provides a set of predefined grouping functions that group traffic using commonly-used criterion (*e.g.,* TCP flow).
- **Shuffle:** Shuffles is similar to Group By except that the number of output branches depends on the number of active cores. The runtime uses the group ID output by the shuffle node to decide the core on which the rest of the processing for the packet will be run. Similar to Group By, NF writers can use both user-defined functions and predefined functions with shuffle nodes. Semantically, the main difference lies in the fact that shuffle outputs are processed on other cores, and the number of outputs is not known at compile time.
- **Merge:** Merge provides a node where separate processing branches can be merged together. All packets entering a merge node exit as a single group.

State Abstraction Modern processors can cheaply provide consistent (serializable) access to data within a core; however, cross-core access comes at a performance cost because of the communication required for cache coherence and the inherent cost of using synchronization primitives such as locks. As a result, NFs are commonly programmed to partition state and avoid such cross-core accesses when possible, or use looser consistency (reducing the frequency of such accesses) when state is not partitionable in this way. Rather than requiring NF writers to partition state and reason about how to implement their desired consistency guarantees, NetBricks provides state abstractions.

Our state abstractions partition the data across cores. Accesses within a core are always synchronized, but we provide several options for other accesses, including (a) no-external-access, *i.e.,* only one core accesses each partition; (b) bounded inconsistency where only one core can write to a partition, but other cores can read these writes within a user supplied bound (specified as number of updates); and (c) strict-consistency where we use traditional synchronization mechanisms to support serializable multi-reader, multi-writer access.

Abstractions for Scheduled Events We also support invocation nodes, which provide a means to run arbitrary UDFs at a given time (or periodically), and can be used to perform tasks beyond packet processing (*e.g.,* collect statistics from a monitoring NF).

3.2 Execution Environment

Next we describe NetBricks's runtime environment, which is responsible for providing isolation between NFs, and NF placement and scheduling.

Isolation As we discuss in §5, container and VM based isolation comes at a significant penalty for simple NFs (for very complex NFs, the processing time inside the NF dominates all other factors, and this is where the efficiency of the NFs built with NetBricks becomes critical). NetBricks therefore takes a different tack and uses software isolation. Previously, Singularity [20] showed that the use of safe languages (*i.e.,* ones which enforce certain type checks) and runtimes can be used to provide memory isolation that is equivalent to what is provided by the hardware memory management unit (MMU) today. NetBricks borrows these ideas and builds on a safe language (Rust) and uses LLVM [28] as our runtime. Safe languages and runtime environments provide four guarantees that are crucial for providing memory isolation in software: (a) they disallow pointer arithmetic, and require that any references acquired by a code is either generated due to an allocation or a function call; (b) they check bounds on array accesses, thus preventing stray memory accesses due to buffer overflows (and underflows); (c) they disallow accesses to null object, thus preventing applications from using undefined behavior to access memory that should be isolated; and (d) they ensure that all type casts are safe (and between compatible objects). Traditionally, languages providing these features (*e.g.,* Java, C#, Go, etc.) have been regarded as being too slow for systems programming.

This situation has improved with recent advances in language and runtime design, especially with the widespread adoption of LLVM as a common optimization backend for compilers. Furthermore, recent work has helped eliminate bounds checks in many common situations [2], and recently Intel has announced hardware support [18] to reduce the overhead of such checks. Finally, until recently most safe languages relied on garbage collection to safely allocate memory. The use of garbage collection results in occasional latency spikes which can adversely affect performance. However, recent languages such as Rust have turned to using reference counting (smart pointers) for heap allocations, leading to predictable latency for applications written in these languages. These developments prompted us to revisit the idea of software isolation for NFs; as we show later in §5, NetBricks achieves throughputs and 99th percentile latency that is comparable with NFs written in more traditional system languages like C.

NFV requires more than just memory isolation; NFV must preserve the semantics of physical networks in the sense that an NF cannot modify a packet once it has been sent (we call this *packet isolation*). This is normally implemented by copying packets as they are passed from NF to NF, but this copying incurs a high performance overhead in packet-processing applications. We thus turn to unique types [13] to eliminate the requirement that packets be copied, while preserving packet isolation.

Unique types, which were originally proposed as a means to prevent data races, disallow two threads from simultaneously having access to the same data. They were designed so that this property could be statically verified at compile time, and thus impose no runtime overhead. We design NetBricks so that calls between NFs are marked to ensure that the sender looses access to the packet, ensuring that only a single NF has access to the packet. This allows us to guarantee that packet isolation holds without requiring any copying. Note that it is possible that some NFs (*e.g.,* IDSes or WAN optimizers) might require access to packet payloads after forwarding packets; in this case the NF is responsible for copying such data.

We refer to the combination of these techniques as Zero-Copy Soft Isolation (ZCSI), which is the cornerstone of NetBricks's execution environment. NetBricks runs as a single process, which maybe assigned one or more cores for processing and one or more NICs for packet I/O. We forward packets between NFs using function calls (*i.e.,* in most cases there are no queues between NFs in a chain, and queuing is done by the receiving NF).

Placement and Scheduling A single NetBricks process is used to run several NFs, which we assume are arranged in several parallel directed graphs – these parallel graphs would be connected to different network interfaces, as might be needed in a multi-tenant scenario where different tenants are handled by different chains of NFs. In addition to the nodes corresponding to the abstractions discussed above, these graphs have special nodes for receiving packets from a port, and sending packets out a NIC. Before execution NetBricks must decide what core is used to run each NF chain. Then, since at any time there can be several nodes in this graph with packets to process, NetBricks must make scheduling decisions about which packet to process next.

For placement, we envision that eventually external management systems (such as E2 [37]) would be responsible for deciding how NFs are divided across cores. At present, to maximize performance we place an entire NF chain on a single core, and replicate the processing graph across cores when scaling. More complex placement policies can be implemented using shuffle nodes, which allow packets to be forwarded across cores.

We use run-to-completion scheduling, *i.e.,* once a packet has entered the NF, we continue processing it until it exits. This then leaves the question of the order in which we let packets enter the NF, and how we schedule events that involve more than one packet. We denote such processing nodes as "schedulable", and these include nodes for receiving packets from a port, Window nodes (which need to schedule their UDF to run when enough data has been collected), and Group By nodes (which queue up packets to be processed by each of the branches). Currently, we use a round-robin scheduling policy to schedule among these nodes (implementing more complex scheduling is left to future work).

4 Implementation

While the previous section presented NetBricks's overall design, here we describe some aspects of its use and implementation.

4.1 Two Example NFs

We use two example NFs to demonstrate how NFs are written in NetBricks. First, in Listing 1 we present a trivial NF that decrements the IP time-to-live (TTL) field and drops any packets with TTL 0. NFs in NetBricks are generally packaged as public functions in a Rust module, and

```
1  pub fn ttl_nf<T: 'static + NbNode>(input: T)
2                        -> CompositionNode {
3    input.parse::<MacHeader>()
4         .parse::<IpHeader>()
5         .transform(box |pkt| {
6              let ttl = pkt.hdr().ttl() - 1;
7              pkt.mut_hdr().set_ttl(ttl);
8         })
9         .filter(box |pkt| {
10             pkt.hdr().ttl() != 0
11        })
12        .compose()
13 }
```

Listing 1: NetBricks NF that decrements TTL, dropping packets with TTL=0.

```
1  // cfg is configuration including
2  // the set of ports to use.
3  let ctx = NetbricksContext::from_cfg(cfg);
4  ctx.queues.map(|p| ttl_nf(p).send(p));
```

Listing 2: Operator code for using the NF in Listing 1

an operator can create a new instance of this NF using the ttl_nf function (line 1), which accepts as input a “source” node. The NF’s processing graph is connected to the global processing graph (*i.e.,* the directed graph of how processing is carried out end-to-end in a NetBricks deployment) through this node. The NF’s processing graph first parses the ethernet (MAC) header from the packet (line 3), and then parses the IP header (line 4). Note that in this case where the IP header begins depends on the contents of the ethernet header and can vary from packet to packet. Once the IP header has been parsed the NF uses the transform operator to decrement each packet’s TTL. Finally, we use the filter operator to drop all packets with TTL 0. The compose operator at the end of this NF acts as a marker indicating NF boundaries, and allows NFs to be chained together. This NF includes no shuffle operators, however by default NetBricks ensures that packets from the same flow are processed by a single core. This is to avoid bad interactions with TCP congestion control. Listing 2 shows how an operator might use this NF. First, we initialize a NetbricksContext using a user supplied configuration (Line 2). Then we create pipelines, such that for each pipeline (a) packets are received from an input queue; (b) received packets are processed using ttl_nf; and (c) packets are output to the same queue. Placement of each pipeline in this case is determined by the core to which a queue is affinitized, which is specified as a part of the user configuration.

Next, in Listing 3 we present a partial implementation of Maglev [9], a load balancer built by Google that was the subject of a NSDI 2016 paper. Maglev is responsible for splitting incoming user requests among a set of backend servers, and is designed to ensure that (a) it can be deployed in a replicated cluster for scalability and fault tolerance; (b) it evenly splits traffic between backends; and (c) it gracefully handles failures, both within the Maglev cluster and among the backends. Maglev uses a novel consistent hashing algorithm (based on a lookup table) to achieve these aims. It however needs to record the mapping between flows and backends to ensure that flows are not rerouted due to failures.

```
1  pub fn maglev_nf<T: 'static + NbNode>(
2                 input: T
3                 backends: &[str],
4                 ctx: nb_ctx,
5                 lut_size: usize)
6                 -> Vec<CompositionNode> {
7    let backend_ct = backends.len();
8    let lookup_table =
9        Maglev::new_lut(ctx,
10             backends,
11             lut_size);
12   let mut flow_cache =
13     BoundedConsistencyMap::<usize, usize>::new();
14
15   let groups =
16     input.shuffle(BuiltInShuffle::flow)
17          .parse::<MacHeader>()
18          .group_by(backend_ct, ctx,
19              box move |pkt| {
20                let hash =
21                  ipv4_flow_hash(pkt, 0);
22                let backend_group =
23                   flow_cache.entry(hash)
24                   .or_insert_with(|| {
25                  lookup_table.lookup(hash)});
26                backend_group
27      });
28     groups.iter().map(|g| g.compose()).collect()
29 }
```

Listing 3: Maglev [9] implemented in NetBricks.

The code in Listing 3 represents the packet processing and forwarding portions of Maglev; our code for generating the Maglev lookup table and consistent hashing closely resemble the pseudocode in Section 3.4 of the paper. The lookup table is stored in a bounded consistency state store, which allows the control plane to update the set of active backends over time. An instance of the Maglev NF is instantiated by first creating a Maglev lookup table (Line 8) and a cache for recording the flow to backend server mappings (Line 12). The latter is unsynchronized (*i.e.,* it is not shared across cores); this is consistent with the description in the Maglev paper. We then declare the NF (starting at line 15); we begin by using a shuffle node to indicate that the NF need all packets within a flow (line 16) to be processed by the same core, then parse the ethernet header, and add a group by node (Line 18). The group by node uses ipv4_flow_hash, a convenience function provided by NetBricks, to extract the flow hash (which is based on both the IP header and the TCP or UDP header of the packet) for the packet. This function is also responsible for ensuring that the packet is actually a TCP or UDP packet (the returned hash is 0 otherwise). The NF then uses this hash to either find the backend previously assigned to this flow (line 24) or assigns a new backend using the lookup table (line 25); this determines the group to which the packet being processed belongs. Finally, the NF returns a vector of composition nodes, where the n^{th} composition node corresponds to the n^{th} backend specified by the operator. The operator can thus forward traffic to each of the backends (or perform further processing) as appropriate. We compare the performance of the NetBricks version of Maglev to Google's reported performance in §5.2.

4.2 Operator Interface

As observed in the previous examples, operators running NetBricks chain NFs together using the same language (Rust) and tools as used by NF authors. This differs from current NF frameworks (*e.g.,* E2, OpenMANO, etc.) where operators are provided with an interface that is distinct from the language used to write network functions. Our decision to use the same interface is for two reasons: (a) it provides many optimization opportunities, in particular we use the Rust compiler's optimization passes to optimize the operator's chaining code, and can use LLVM's link-time optimization passes [28] to perform whole-program optimization, improving performance across the entire packet processing pipeline; and (b) it provides an easy means for operators to implement arbitrarily complicated NF chaining and branching.

4.3 Implementation of Abstractions

We now briefly discuss a few implementation details for abstractions in NetBricks. First, packet processing abstractions in NetBricks are lazy; *i.e.,* they do not perform computation until the results are required for processing. For example, parse nodes in NetBricks perform no computation until a transform, filter, group by, or similar node (*i.e.,* a node with a UDF that might access the packet header or payload) needs to process a packet. Secondly, as is common in high-performance packet processing, our abstractions process batches of packets at a time. Currently each of our abstractions implements batching to maximize common-case performance, in the future we plan on looking at techniques to choose the batching technique based on both the UDF and abstraction.

4.4 Execution Environment

The NetBricks framework builds on Rust, and we use LLVM as our runtime. We made a few minor modifications to the default Rust nightly install: we changed Cargo (the Rust build tool) to pass in flags that enabled machine specific optimizations and the use of vector instructions for fast memory access; we also implemented a Rust lint that detects the use of unsafe pointer arithmetic inside NFs, and in our current implementation we disallow building and loading of NF code that does not pass this lint. Beyond these minor changes, we found that we could largely implement our execution environment using the existing Rust toolchain. In the future we plan to use tools developed in the context of formal verification efforts like RustBelt [7] to (a) statically verify safety conditions in binary code (rather than relying on the Lint tool) and (b) eliminate more of the runtime checks currently performed by NetBricks.

5 Evaluation

5.1 Setup

We evaluate NetBricks on a testbed of dual-socket servers equipped with Intel Xeon E5-2660 CPUs, each of which has 10 cores. Each server has 128GB of RAM, which is divided equally between the two sockets. Each server is also equipped with an Intel XL710 QDA2 40Gb NIC. For our evaluation we disabled hyper-threading and adjusted the power settings to ensure that all cores ran at a constant 2.6GHz[2]. We also enabled hardware virtualization features including Intel VT. These changes are consistent with settings recommended for NFV applications. The servers run Linux kernel 4.6.0-1 and NetBricks uses DPDK version 16.04 and the Rust nightly version. For our tests we relied on two virtual switches (each configured as recommended by authors): OpenVSwitch with DPDK (OVS DPDK) [15], the de-facto virtual switch used in commercial deployments, and SoftNIC [17], a new virtual switch that has been specifically optimized for NFV use cases [37].

We run VMs using KVM; VMs connect to the virtual switch using DPDK's `vhost-user` driver. We run containers using Docker in privileged mode (as required by DPDK [8]), and connect them to the virtual switch using DPDK's ring PMD driver. By default, neither OpenVSwitch nor SoftNIC copy packets when using the ring PMD driver and thus do not provide packet isolation (because an NF can modify packets it has already sent). For most of our evaluation we therefore modify these switches to copy packets when connecting containers. However, even with this change, our approach (using DPDK's ring PMD driver) outperforms the commonly recommended approach of connecting containers with virtual switches using `veth` pairs (virtual ethernet devices that connect through the kernel). These devices entail a copy in the kernel, and hence have significantly worse performance than the ring based connections we use. Thus, the performance we report are a strict *upper bound* on can be achieved using containers safely.

For test traffic, we use a DPDK-based packet generator that runs on a separate server equipped with a 40Gb NIC and is directly connected to the test server without any in-

[2]In particular we disabled C-state and P-state transitions, isolated CPUs from the Linux scheduler, set the Linux CPU QoS feature to maximize performance, and disabled uncore power scaling.

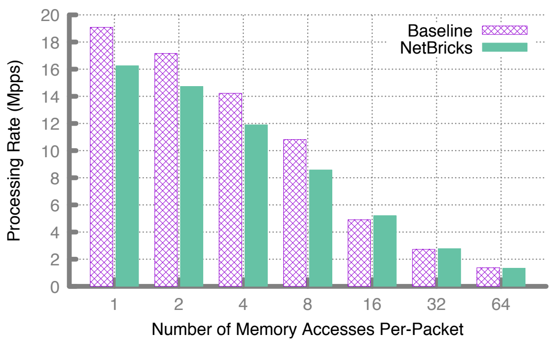


Figure 1: Throughput achieved by a NetBricks NF and an NF written in C using DPDK as the number of memory accesses in a large array grows.

tervening switches.The generator acts as traffic source and sink and we report the throughput and latency measured at the sink. For each run we measure the maximum throughput we can achieve with zero packet loss, and report the median taken across 10 runs.

5.2 Building NFs

5.2.1 Framework Overheads

We begin by evaluating the overheads imposed by NetBricks' programming model when compared to baseline NFs written more traditionally using C and DPDK. To ensure that we measure only framework overheads we configure the NIC and DPDK in an identical manner for both NetBricks and the baseline NF.

Overheads for Simple NFs As an initial sanity check, we began by evaluating overheads on a simple NF (Listing 1) that on receiving a packet, parses the packet until the IP header, then decrements the packet's IP time-to-live (TTL) field, and drops any packets whose TTL equals 0. We execute both the NetBricks NF and the equivalent C application on a single core and measure throughput when sending 64 byte packets. As expected, we find that the performance for the two NFs is nearly identical: across 10 runs the median throughput for the native NF is 23.3 million packet per-second, while NetBricks achieves 23.2 million packets per second. In terms of latency, at 80% load, the 99^{th} percentile round trip time for the native NF is 16.15μs, as compared to 16.16μs for NetBricks.

Overheads for Checking Array Bounds Our use of a safe language imposes some overheads for array accesses due to the cost of bounds checking and such checks are often assumed to be a dominant source of overhead introduced by safe languages.[3] While these checks can be eliminated statically in some cases (*e.g.,* where bounds can be placed on the index statically), this is not always possible. We measured the impact of these checks using a network function that updates several cells in a 512KB array while processing each packet. The set of cells to be updated depends on the UDP source port number of the packet being processed, making it impossible to eliminate array bounds checks. We compare the overheads for our implementation in NetBricks to a baseline NF written in C (using DPDK for packet I/O), and behaving identically. In both cases we use a single-core and use packets with randomly assigned UDP source ports. Figure 1 shows the throughput achieved by each NF as the number of memory accesses per packet is increased. When processing a packet necessitates a single memory access, NetBricks imposes a 20% overhead compared to the baseline. We see that this performance overhead remains for a small number (1-8) of accesses per packet. However, somewhat counter-intuitively, with 16 or higher accesses per packet, the performance overhead of our approach drops; this is because, at this point, the number of cache misses grows and the performance impact of these misses dominates that from our bounds checks.

To test the impact of this overhead in a more realistic application we implemented a longest prefix match (LPM) lookup table using the DIR-24-8 algorithm [16] in Rust, and built a NetBricks NF that uses this data structure to route IP packets. We compare the performance of this NF to one implemented in C, which uses the DIR-24-8 implementation included with DPDK [24]. Lookups using this algorithm require between 1 and 2 array accesses per packet. For our evaluation we populated this table with 16000 random rules. We find that NetBricks can forward 15.73 million packet per second, while the native NF can forward 18.03 million packets per second (so the NetBricks NF is 14% slower). We also measure the 99^{th} percentile round trip time at 80% load (*i.e.,* the packet generator was generating traffic at 80% of the 0-loss rate), this value indicates the per-packet latency for the NF being tested. The $99th$ percentile RTT for NetBricks was 18.45μs, while it was 16.15μs for the native NF, which corresponds to the observed difference in throughputs.

5.2.2 Generality of Programming Abstractions

To stress test the generality of our programming abstractions, we implemented a range of network functions from the literature:

- Firewall: is based on a simple firewall implemented in Click [5]; the firewall performs a linear scan of an access control list to find the first matching entry.
- NAT: is based on MazuNAT [41] a Click based NAT implemented by Mazu Networks, and commonly used in academic research.

[3] Null-checks and other safety checks performed by the Rust runtime are harder to separate out; however, these overheads are reflected in the overall performance we report below.

NF	NetBricks	Baseline
Firewall	0.66	0.093
NAT	8.52	2.7
Sig, Matching	2.62	0.983
Monitor	5	1.785

Table 1: Throughputs for NFs implemented using NetBricks as compared to baseline from the literature.

- Signature Matching: a simple NF similar to the core signature matching component of the Snort intrusion prevention system [42].
- Monitor: maintains per-flow counters similar to the monitor module found in Click and commonly used in academic research [43]
- Maglev: as described in § 4, we implemented a version of the Maglev scalable load-balancer design [9].

In Table 1, we report the per-core throughput achieved by the first four applications listed above, comparing our NetBricks implementation and the original system on which we based on our implementation. We see that our NetBricks implementations often outperform existing implementations – *e.g.,* our NAT has approximately 3× better performance than MazuNAT [41]. The primary reason for this difference is that we incorporate many state-of-the-art optimizations (such as batching) that were not implemented by these systems.

In the case of Maglev, we do not have access to the source code for the original implementation and hence we recreate a test scenario similar to that corresponding to Figure 9 in [9] which measures the packet processing throughput for Maglev with different kinds of TCP traffic. As in [9], we generate short-lived flows with an average of 10 packets per flow and use a table with 65,537 entries (corresponding to the small table size in [9]). Our test server has a 40Gbps link and we measure throughput for (min-size) 64B packets. Table 2 shows the throughput achieved by our NetBricks implementation for increasing numbers of cores (in Mpps), together with comparable results reported for the original Maglev system in [9]. We see that our NetBricks implementation offers between 2.9× and 3.5× better performance than reported in [9]. The median latency we observed in this case was 19.9μS while 99th percentile latency was 32μS. We note however that (a) we ran on different hardware; and (b) we did not have access to the base implementation and hence comment on parity. Therefore these numbers are not meant to indicate that our performance is better, just that NetBricks can achieve comparable results as obtained by a hand tuned NF.

Our point here is not that NetBricks will outperform highly optimized native implementations; instead, our results merely suggest that NetBricks can be used to implement a wide variety of NFs, and that these implementations

# of Cores	NetBricks Impl.	Reported
1	9.2	2.6
2	16.7	5.7
3	24.5	8.2
4	32.24	10.3

Table 2: Throughputs for the NetBricks implementation of Maglev (NetBricks) when compared to the reported throughput in [9] (Reported) in millions of packets per second (MPPS).

are both simpler than the native implementations (*e.g.,* our Maglev implementation is 150 lines of code) and roughly comparable in performance.

5.3 Execution Environment

NetBricks exploits the isolation properties of safe languages and runtime checks to avoid the costs associated with crossing process and/or core boundaries. We first quantify these savings in the context of a single NF and then evaluate how these benefits accrue as the length of a packet's NF chain increases. Note that these crossing costs are only important for simple NFs; once the computational cost of the NF becomes the bottleneck, then our execution environment becomes less important (though NetBricks's ability to simply implement high-performance NFs becomes more important).

5.3.1 Cost of Isolation: Single NF

We evaluate the overhead of using VMs or containers for isolation and compare the resultant performance to that achieved with NetBricks. We first consider the simplest case of running a single test NF (which is written using NetBricks) that swaps the source and destination ethernet address for received packets and forwards them out the same port. The NetBricks NF adds no additional overhead when compared to a native C NF, and running the same NF in all settings (VM, containers, NetBricks) allows us to focus on the cost of isolation.

The setup for our experiments with containers and VMs is shown in Figure 2: a virtual switch receives packets from the NIC, these packets are then forwarded to the NF which is running within a VM or container. The NF processes the packet and sends it back to the vSwitch, which then sends it out the physical NIC. Our virtual switches and NFs run on DPDK and rely on polling. We hence assign each NF its own CPU core and assign two cores to the switch for polling packets from the NIC and the container.[4] Isolation introduces two sources of overheads: overheads from cache and context switching costs associated with crossing process (and in our case core) boundaries, and overheads

[4]This configuration has been shown to achieve better performance than one in which the the switch and NFs share a core [35]. Our own experiments confirm this, we saw as much as 500% lower throughput when cores were shared.

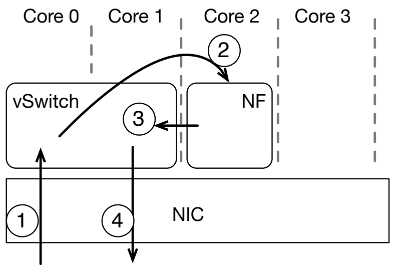


Figure 2: Setup for evaluating single NF performance for VMs and containers.

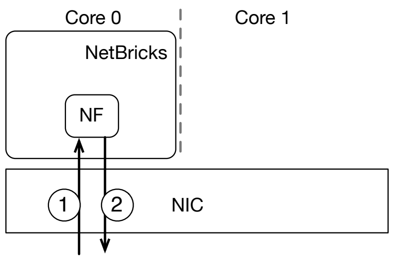


Figure 3: Setup for evaluating single NF performance using NetBricks.

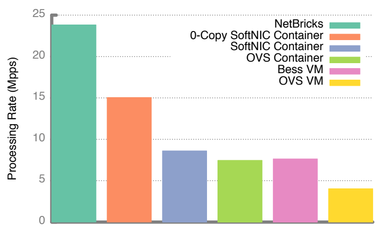


Figure 4: Throughput achieved using a single NF running under different isolation environments.

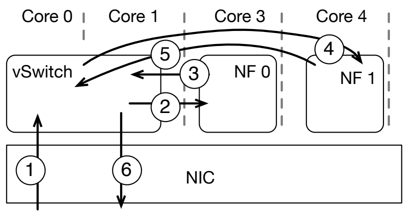


Figure 5: Setup for evaluating the performance for a chain of NFs, isolated using VMs or Containers.

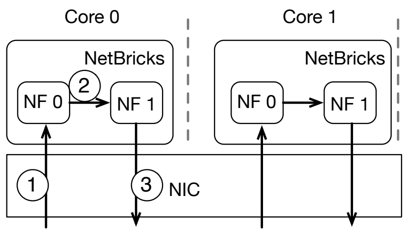


Figure 6: Setup for evaluating the performance for a chaining of NFs, running under NetBricks.

from copying packets. To allow us to analyze these effects separately we include in our results a case where SoftNIC is configured to send packets between containers without copying (0-copy SoftNIC Container), even though this violates our desired packet isolation property. We compare these results to NetBricks running in the setup shown in Figure 3. In this case NetBricks is responsible for receiving packets from the NIC, processing them using the NF code and then sending them back out. We run NetBricks on a single core for this evaluation.

Figure 4 shows the throughput achieved for the different isolation scenarios when sending 64B minimum sized packets. Comparing the 0-copy SoftNIC throughput against NetBricks's throughput, we find that just crossing cores and process isolation boundaries results in performance degradation of over 1.6× when compared to NetBricks (this is despite the fact that our NetBricks results used fewer cores overall; 1 core for NetBricks vs. 3 in the other cases). When packets are copied (SoftNIC Container) throughput drops further and is 2.7× worse than NetBricks. Generally the cost for using VMs is higher than the cost for using Containers; this is because Vhost-user, a virtualized communication channel provided by DPDK for communicating with VMs imposes higher overheads than the ring based communication channel we use with containers.

The previous results (Figure 9) focused on performance with 64B packets, and showed that as much as 50% of the overhead in these systems might be due to copying packets. We expect that this overhead should increase with larger packets, hence we repeated the above tests for 1500B packets and found that the per-packet processing time (for those scenarios that involve copying packets) increased by approximately 15% between 64B and 1500B packets (the small size of the increase is because the cost of allocation dominates the cost of actually copying the bits).

5.3.2 Cost of Isolation: NF Chains

Next, we look at how performance changes when each packet is handled by a *chain* of NFs. For simplicity, we generate chains by composing multiple instances of a single test NF; *i.e.,* every NF in the chain is identical and we only vary the length of the chain. Our test NF performs the following processing: on receiving a packet, the NF parses the ethernet and IP header, and then decrement the time-to-live (TTL) field in the IP header. The NF drops any packets where the TTL is 0.

We use the setup shown in Figure 5 to measure these overheads when using VMs and containers. As before, we assign the virtual switch two cores, and we place each VM or container on a separate core. We evaluate NetBricks using the setup shown in Figure 6. We ran NetBricks in two configurations: (a) one where NetBricks was run on a single core, and (b) another where we gave NetBricks as many cores as the chain length; in the later case NetBricks uses as many cores as the container/VM runs.

In Figure 7 we show the throughput as a function of increasing chain length. We find that NetBricks is up to 7× faster than the case where containers are connected using SoftNIC and up to 11× faster than the case where VMs are connected using SoftNIC. In fact NetBricks is faster

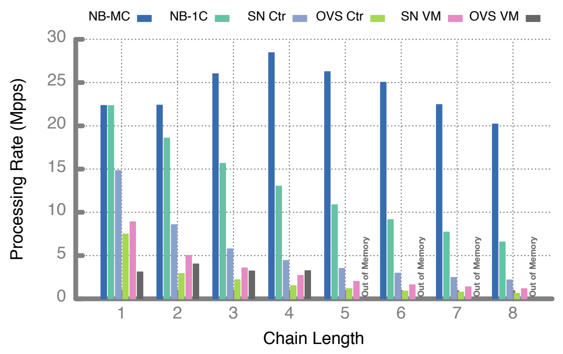


Figure 7: Throughput with increasing chain length when using 64B packets. In this figure NB-MC represents NetBricks with multiple cores, NB-1C represents NetBricks with 1 core.

even when run on a single core, we observe that it provides 4× higher throughput than is achieved when containers are connected through SoftNIC, and up to 6× higher throughput when compared to the case where VMs are connected using SoftNIC. Furthermore, by comparing to the 0-copy SoftNIC case, we find that for 64B packets copying can result in a performance drop of up to 3×. Finally, observe that there is a dip in NetBricks's performance with multiple cores once the chain is longer than four elements. This is because in our setup I/O becomes progressively more expensive as more cores access the same NIC, and with more than 4 parallel I/O threads this cost dominates any improvements from parallelism. We believe this effect is not fundamental, and is a result of our NIC and the current 40Gbps driver in DPDK. NetBricks's performance benefits are even higher when we replace SoftNIC with OpenVSwitch.[5]

The above results are for 64B packets; as before, we find that while copying comes at a large fixed cost (up to 3× reduction in throughput), increasing packet sizes only results in an approximately 15% additional degradation. Finally, we also measured packet processing latency when using NetBricks, containers and VMs; Figure 8 shows the 99th percentile round trip time at 80% of the maximum sustainable throughput as a metric for latency.

Effect of Increasing NF Complexity Finally, we analyze the importance of our techniques for more complex NFs. We use cycles required for processing each packet as a proxy for NF complexity. We reuse the setup for single NF evaluations (Figure 2, 3), but modify the NF so that it busy loops for a given number of cycles after modifying the packet, allowing us to vary the per-packet processing time. Furthermore, note that in the case where VMs or containers the setup itself uses 3 cores (1 for the NF and 2

[5]We were unable to run experiments with more than four VMs chained together using OpenVSwitch because we ran out of memory in our configuration.

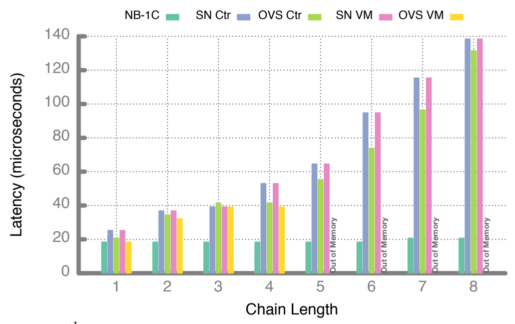


Figure 8: 99th percentile RTT for 64B packets at 80% load as a function of chain length.

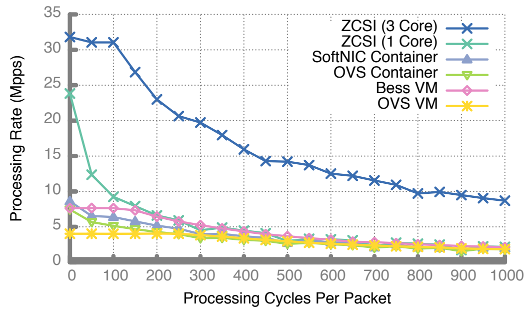


Figure 9: Throughput for a single NF with increasing number of cycles per-packet using different isolation techniques.

for the vSwitch). Therefore, for this evaluation, in addition to measuring performance with NetBricks on 1 core, we also measure performance when NetBricks is assigned 3 cores (equalizing resources across the cases).

Figure 9 shows the throughput as a function of per-packet processing requirements (cycles). As expected, as we increase NF complexity, packet processing time starts to be the dominant factor for determining performance, and our runtime improvements have minimal effect once an NF needs more than 300 cycles per packet. This reduction in benefits when NF processing demands dominate also applies to fast packet processing libraries such as DPDK. Note however that the gains when NetBricks is given as many cores as the traditional approaches (three) continue to be significant even when NFs need more than 1000 cycles per packet. Thus, NetBricks' approach to isolation provides better performance per unit of allocated resource when compared to current approaches.

6 Related Work

The works most closely related to NetBricks' programming model are Click and Snabb switch [14]. We have compared NetBricks and Click throughout the paper, do not provide further discussion here. Recent extensions to Click, *e.g.,* NBA [26] and ClickNP [29], have looked at how to implement optimized Click elements through the use of GPUs (NBA) and FPGAs (ClickNP). While offloading function-

Framework	Memory Isolation	Packet Isolation	Overheads
xOMB [1]	✗	✗	Low (function call)
CoMB [43]	✗	✗	Low (function call)
NetVM [21]	✓	✗	Very high (VM)
ClickOS [32]	✓	✓	High (lightweight VM)
HyperSwitch [40]	✓	✓	Very high (VM)
mSwitch [19]	✓	✓	Very high (VM)
NetBricks	✓	✓	Low (function call)

Table 3: A comparison with other NFV frameworks.

ality to such devices can yield great performance improvements, this is orthogonal to our work. Adding the use of offloads in NetBricks is left to future work. Snabb provides the same programming model as Click but uses Lua [22] instead of C++ for programming, which allows the use of a high-level language but without actually raising the level of abstraction (in terms of having the programmer deal with all the low-level packet-handling issues).

There has also been a long line of work on developing network applications on specialized networking hardware including NPUs [46], FPGAs [33] and programmable switches [4]. Recent work including P4 [3] and Packet Transactions [45] have looked at providing high level programming tools for such hardware. Our work focuses on network programming for general purpose CPUs and is complementary to this work.

In terms of NetBricks' execution model, work on library operating systems (*e.g.,* MirageOS [31] and Drawbridge [39]) has decreased VM resource overheads and improved VM performance by reducing the amount of code run within each VM and improving the hypervisor. While these projects have provided substantial performance improvements, they do not eliminate the isolation overheads we focus on here nor do they address how to perform efficient I/O in this environment.

As we have noted previously, our execution model is closely related to software-isolated processes (SIPs) proposed by Singularity [20]. The main difference is that our work focuses on a single application domain – network functions – where inter-NF communication is common and greatly benefits from the use of software isolation. Furthermore, Singularity was designed as a general purpose, microkernel-based operating system, and focused on providing an efficient general implementation for application developers. As a result Singularity's design choices – *e.g.,* the use of a garbage collected language, communication through an exchange heap and queued channel, etc. – are not optimized for the NFV use case.

Other work has proposed a variety of execution frameworks specific to NFV [1,19,21,32,40,43]. We can broadly divide these frameworks into two groups: Click-like frameworks that run all NFs in a single process without isolation, and VM-based frameworks. We present a comparison of these frameworks and NetBricks in Table 3. As shown, only NetBricks provides both isolation and low overheads. Finally, In-Net [47] has looked at providing traffic isolation for NFs in a network, and is orthogonal to our work.

Several attempts have also been made to offload vSwitch functionality to NIC hardware. For example, FasTrak [34] advocates using hardware virtualization (SR-IOV [6]) and built-in switching capabilities of commodity NICs to interconnect VMs. This approach eliminates the cost of copying packets in software by using hardware DMA. However, I/O bus bandwidth is an order-of-magnitude lower (a few GB/s) than cache and memory bandwidth (10s-100s of GB/s), and this limits the number of packets that can be transmitted in parallel and thus reduces the throughput that can be achieved. Offloading switching to hardware also limits flexibility in how packets are steered across NFs; *e.g.,* Intel's 10 G NICs only support basic L2 switching.

IO-Lite [36], Container Shipping [38], and work done for Solaris [25] have looked at solutions for implementing zero-copy I/O. IO-Lite provided zero-copy isolation by making buffers immutable. This necessitates creating a new buffer on any write (similar to copy-on-write techniques) and would therefore incur performance degradation when modifications are required. Container shipping and the Solaris approach unmap pages from the sending process to provide zero-copy isolation. Page table modifications require a trap into the kernel, and come at a significant penalty [48]. By contrast our implementation of 0-copy I/O imposes no runtime overheads.

7 Conclusion

As can be seen from our brief review of related work, NetBricks is the only approach that enables developers to write in high-level abstractions (thereby easing development) while maintaining good performance and memory/packet isolation. We are continuing to explore the limits of NetBricks's generality – by implementing new NFs – and increase the range of NetBricks's low-level optimizations, some of which are currently rather primitive. In service of these goals, we have also made NetBricks and our examples available to the community at `netbricks.io`.

8 Acknowledgment

We thank our shepherd George Porter and the anonymous reviewers for their comments. We also thank Ion Stoica, Amin Tootoonchian and Shivaram Venkatraman for their helpful feedback, which influenced both the design of our system and the contents of this paper. This work was funded in part by a grant from Intel Corporation, and by NSF awards 1216073 and 1420064.

References

[1] J. W. Anderson, R. Braud, R. Kapoor, G. Porter, and A. Vahdat. xOMB: Extensible Open Middleboxes with Commodity Servers. In *ANCS*, 2012.

[2] R. Bodík, R. Gupta, and V. Sarkar. ABCD: Eliminating Array Bounds Checks on Demand. In *PLDI*, 2000.

[3] 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 Computer Communication Review*, 44(3):87–95, 2014.

[4] 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.

[5] M. Dobrescu, K. Argyraki, and S. Ratnasamy. Toward Predictable Performance in Software Packet-Processing Platforms. In *Presented as part of the 9th USENIX Symposium on Networked Systems Design and Implementation (NSDI 12)*, 2012.

[6] Y. Dong, X. Yang, L. Xiaoyong, J. Li, H. Guan, and K. Tian. High Performance Network Virtualization with SR-IOV. In *IEEE HPCA*, 2012.

[7] D. Dreyer. RustBelt: Logical Foundations for the Future of Safe Systems Programming. http://plv.mpi-sws.org/rustbelt/ (Retrieved 05/05/2016), 2015.

[8] J. Eder. Can you run DPDK in a Container. Redhat Blog http://goo.gl/UBdZpL, 2015.

[9] D. E. Eisenbud, C. Yi, C. Contavalli, C. Smith, R. Kononov, E. Mann-Hielscher, A. Cilingiroglu, B. Cheyney, W. Shang, and J. D. Hosein. Maglev: A Fast and Reliable Software Network Load Balancer. In *NSDI*, 2016.

[10] ETSI. Network Functions Virtualisation. Retrieved 07/30/2014 http://portal.etsi.org/NFV/NFV_White_Paper.pdf.

[11] L. Foundation. OPNFV. https://www.opnfv.org/, 2016.

[12] A. Gember, A. Krishnamurthy, S. S. John, R. Grandl, X. Gao, A. Anand, T. Benson, A. Akella, and V. Sekar. Stratos: A Network-Aware Orchestration Layer for Middleboxes in the Cloud. http://arxiv.org/abs/1305.0209, 2013.

[13] C. S. Gordon, M. J. Parkinson, J. Parsons, A. Bromfield, and J. Duffy. Uniqueness and Reference Immutability for Safe Parallelism. In *OOPSLA*, 2012.

[14] L. Gorrie. SNABB Switch. https://goo.gl/8ox9kE retrieved 07/16/2015.

[15] J. Gross. The Evolution of OpenVSwitch. http://goo.gl/p7QVek retrieved 07/13/2015, 2014. Talk at LinuxCon.

[16] P. Gupta, S. Lin, and N. McKeown. Routing Lookups in Hardware at Memory Access Speeds. In *INFOCOM*, 1998.

[17] S. Han, K. Jang, A. Panda, S. Palkar, D. Han, and S. Ratnasamy. SoftNIC: A Software NIC to Augment Hardware. Technical Report UCB/EECS-2015-155, EECS Department, University of California, Berkeley, May 2015.

[18] D. Hansen. Intel Memory Protection Extensions (Intel MPX) for Linux*. https://01.org/blogs/2016/intel-mpx-linux retrieved 05/07/2016, 2016.

[19] M. Honda, F. Huici, G. Lettieri, and L. Rizzo. mSwitch: A Highly-Scalable, Modular Software Switch. In *SOSR*, 2015.

[20] G. C. Hunt and J. R. Larus. Singularity: Rethinking the Software Stack. *ACM SIGOPS Operating Systems Review*, 41(2):37–49, 2007.

[21] J. Hwang, K. Ramakrishnan, and T. Wood. NetVM: High Performance and Flexible Networking using Virtualization on Commodity Platforms. *Network and Service Management, IEEE Transactions on*, 12(1):34–47, 2015.

[22] R. Ierusalimschy, L. H. de Figueiredo, and W. Celes Filho. Lua–an Extensible Extension Language. In *Software: Practice & Experience*, 1995.

[23] Intel. Data Plane Develpment Kit. http://dpdk.org/, 2016.

[24] Intel. DPDK: rte_table_lpm.h reference. http://goo.gl/YBS4UO retrieved 05/07/2016, 2016.

[25] Y. A. Khalidi and M. N. Thadani. An Efficient Zero-Copy I/O Framework for Unix. *Sum Mircrosystems Laboratories, Inc. Tech Report*, 1995.

[26] J. Kim, K. Jang, K. Lee, S. Ma, J. Shim, and S. B. Moon. NBA (Network Balancing Act): A High-Performance Packet Processing Framework for Heterogeneous Processors. In *EuroSys*, 2015.

[27] 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.

[28] C. Lattner and V. Adve. LLVM: A compilation framework for lifelong program analysis & transformation. In *Code Generation and Optimization*. IEEE, 2004.

[29] B. Li, K. Tan, L. L. Luo, Y. Peng, R. Luo, N. Xu, Y. Xiong, and P. Cheng. ClickNP: Highly flexible and High-performance Network Processing with Reconfigurable Hardware. In *SIGCOMM*, 2016.

[30] D. Lopez. OpenMANO: The Dataplane Ready Open Source NFV MANO Stack. In *IETF Meeting Proceedings, Dallas, Texas, USA*, 2015.

[31] A. Madhavapeddy, R. Mortier, C. Rotsos, D. Scott, B. Singh, T. Gazagnaire, S. Smith, S. Hand, and J. Crowcroft. Unikernels: Library operating systems for the cloud. In *ASPLOS*, 2013.

[32] J. Martins, M. Ahmed, C. Raiciu, V. Olteanu, M. Honda, R. Bifulco, and F. Huici. ClickOS and the Art of Network Function Virtualization. In *NSDI*, 2014.

[33] J. Naous, G. Gibb, S. Bolouki, and N. McKeown. NetFPGA: Reusable Router Architecture for Experimental Research. In *Workshop on Programmable routers for extensible services of tomorrow*, 2008.

[34] R. Niranjan Mysore, G. Porter, and A. Vahdat. FasTrak: Enabling Express Lanes in Multi-Tenant Data Centers. In *CoNEXT*, 2013.

[35] OpenVSwitch. Using Open vSwitch with DPDK. https://github.com/openvswitch/ovs/blob/master/INSTALL.DPDK.md, 2016.

[36] V. S. Pai, P. Druschel, and W. Zwaenepoel. IO-Lite: A Unified I/O Buffering and Caching System. *ACM Transactions on Computer Systems (TOCS)*, 18(1):37–66, 2000.

[37] S. Palkar, C. Lan, S. Han, K. Jang, A. Panda, S. Ratnasamy, L. Rizzo, and S. Shenker. E2: A Framework for NFV Applications. In *SOSP*, 2015.

[38] J. Pasquale, E. Anderson, and P. K. Muller. Container Shipping: Operating System Support for I/O-intensive Applications. *Computer*, 27(3):84–93, 1994.

[39] D. E. Porter, S. Boyd-Wickizer, J. Howell, R. Olinsky, and G. C. Hunt. Rethinking the Library OS from the Top Down. In *ASPLOS*, 2011.

[40] K. K. Ram, A. L. Cox, M. Chadha, S. Rixner, T. W. Barr, R. Smith, and S. Rixner. Hyper-Switch: A Scalable Software Virtual Switching Architecture. In *USENIX ATC*, 2013.

[41] Riverbed. Mazu Networks. http://goo.gl/Y6aeEg, 2011.

[42] M. Roesch et al. Snort: Lightweight Intrusion Detection for Networks. In *LISA*, 1999.

[43] V. Sekar, N. Egi, S. Ratnasamy, M. K. Reiter, and G. Shi. Design and Implementation of a Consolidated Middlebox Architecture. In *NSDI*, 2012.

[44] J. Sherry, S. Hasan, C. Scott, A. Krishnamurthy, S. Ratnasamy, and V. Sekar. Making Middleboxes Someone Else's Problem: Network Processing as a Cloud Service. In *SIGCOMM*, 2012.

[45] A. Sivaraman, M. Budiu, A. Cheung, C. Kim, S. Licking, G. Varghese, H. Balakrishnan, M. Alizadeh, and N. McKeown. Packet Transactions: High-level Programming for Line-Rate Switches. In *SIGCOMM*, 2016.

[46] H. Song. Protocol-Oblivious Forwarding: Unleash the Power of SDN Through a Future-Proof Forwarding Plane. In *HotSDN*, 2013.

[47] R. Stoenescu, V. A. Olteanu, M. Popovici, M. Ahmed, J. Martins, R. Bifulco, F. Manco, F. Huici, G. Smaragdakis, M. Handley, and C. Raiciu. In-Net: In-Network Processing for the Masses. In *EuroSys*, 2015.

[48] L. Torvalds. Linux Page Fault Daemon Performance. Google+ https://plus.google.com/+LinusTorvalds/posts/YDKRFDwHwr6, 2014.

Efficient Network Reachability Analysis using a Succinct Control Plane Representation

Seyed K. Fayaz   Tushar Sharma   Ari Fogel*
Ratul Mahajan†   Todd Millstein‡   Vyas Sekar   George Varghese‡

CMU   **Intentionet*   †*Microsoft Research*   ‡UCLA

Abstract— To guarantee network availability and security, operators must ensure that their reachability policies (e.g., A can or cannot talk to B) are correctly implemented. This is a difficult task due to the complexity of network configuration and the constant churn in a network's environment, e.g., new route announcements arrive and links fail. Current network reachability analysis techniques are limited as they can only reason about the current "incarnation" of the network, cannot analyze all configuration features, or are too slow to enable exploration of many environments. We build ERA, a tool for efficient reasoning about network reachability. Instead of reasoning about individual incarnations of the network, ERA directly reasons about the network "control plane" that generates these incarnations. We address key expressiveness and scalability challenges by building *(i)* a succinct model for the network control plane (i.e., various routing protocols and their interactions), and *(ii)* a repertoire of techniques for scalable (taking a few seconds for a network with > 1000 routers) exploration of this model. We have used ERA to successfully find both known and new violations of a range of common intended polices.

1 Introduction

Network operators need to ensure the correct behavior of their networks. Violations of intended reachability policies (e.g., "Can A talk to B?") can compromise availability, security, and performance of the network. This risk has inspired the field of network verification, which aims to enable operators to systematically reason about their networks [39].

Reasoning about a network is hard, as a real network is in a perpetual churn: route advertisements arrive, links fail, and routers need to be taken offline for maintenance. Nonetheless, an operator needs assurances on the network behaviors because a policy violation may be latent and occur only in a certain future *incarnation* (e.g., a specific route advertisement from a peering network may cause disconnection between A and B [6, 11]). Unfortunately, today operators do not have proper tools for efficient reasoning about the network in different environments.

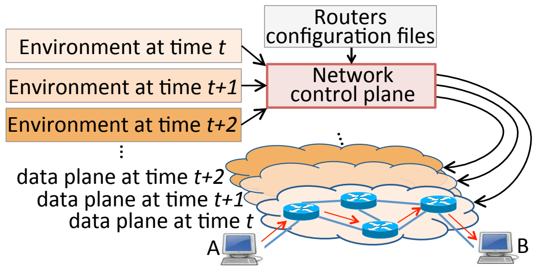


Figure 1: Reachability behavior of a network (e.g., A can talk to B) is determined by its data plane, which, in turn, is the current incarnation of the control plane.

To highlight this challenge, it is useful to consider prior work on network verification. A network is composed of a control plane, which configures the behavior of the data plane, which in turn, is in charge of forwarding actual packets (see Figure 1). The control plane, therefore, can be thought of as a program that takes configuration files and the current network environment (i.e., route advertisements) and generates a data plane. The data plane is conceptually a program that takes a packet and its location (i.e., a router port) as input and outputs a packet at a different location. As a result, if we rest our analysis on the data plane (e.g., Veriflow [29], HSA [28], NOD [36]) and verify its behavior over its inputs (i.e., packets), we are inherently able to reason about only the current incarnation of the control plane (i.e., the current data plane), and cannot say anything about the network behavior under a different environment.

While there is prior work on bug-finding and verification for the control plane, it suffers from critical limitations. Some tools focus on a single routing protocol (e.g., BGP for Bagpipe [41] and rcc [18]) or a limited set of routing protocol features (e.g., ARC [21]). They can thus not capture the behavior of the entire control plane that often uses multiple routing protocols and sophisticated features [22,31,38]. On the other hand, Batfish [19] analyzes the entire control plane in the context of a given environment, but it does so by simulating the behavior of individual routing protocols to compute the resulting data plane. This simulation is expensive (see §9.2), which makes it prohibitive to iteratively use Batfish to analyze the impact of many environments.

What is critically missing today is the ability to efficiently find network reachability bugs across multiple possible environments. (§3 motivates this need using real-world examples.) Doing so requires reasoning about network reachability directly at the control plane level, without explicitly computing the data plane that manifests in each environment. Such reasoning is challenging due to the complexity of the control plane, which involves various routing protocols (e.g., BGP, OSPF, RIP) each with its own intricacies (e.g., selecting best route to a destination prefix is different for BGP and OSPF) and cross-protocol interactions (e.g., route redistribution [32]).

We address these challenges in a tool called ERA (efficient reachability analysis) by employing several synergistic ideas. First, we design a unified control plane model that succinctly captures the key behaviors of various routing protocols. In this model, a router is viewed as a function that accepts a route announcement as input and produces a set of route announcements for its neighbors. Second, we use binary decision diagrams (BDDs) [30] to compactly represent the route announcements that constitute a user-specified environment. Third, we shrink the BDD representation of route announcements by identifying equivalence classes of announcements that are treated identically by the given network [42]. Each equivalence class is given an integer index, and the reachability analysis is transformed to arithmetic operations directly on sets of these indices. Consequently, we take advantage of vectorized instruction sets on commodity CPUs for fast computation of these set operations (§6).

ERA can be used to identify bugs in reachability policies of the form "A can talk to B" as well as a wide range of common policies that are expressible in terms of reachability relationships, such as valley-free routing and blackhole-freeness (§7). Our implementation of ERA is available as an open source and extensible toolkit to which new kinds of analysis can be added (§8).

We evaluate the utility of ERA in a range of real and synthetic scenarios (§9.1). Across all scenarios, it successfully finds both new and known reachability violations, which were otherwise hard to find using the state of the art techniques. We also evaluate the scalability of ERA and find that it can handle a network with over 1,600 routers in 6 seconds. Our evaluations show that our control plane modeling and exploration techniques yield significant speedup.

2 Related Work

There are several strands of related prior work.

Data plane analysis: Verifying the reachability behavior of the data plane has been widely studied (e.g., Anteater [37], Veriflow [29], HSA [28], NOD [36]). The result from such verification, however, is valid only for the specific data plane being analyzed. There has also been extensive work on testing the data plane (e.g., ATPG [43], Pingmesh [26]). Data plane verification and testing is fundamentally limited, as a network is in a constant churn, which manifests itself as different data planes. For example, a single route advertisement can dramatically change the network behavior (e.g., see [11]).

Control plane analysis: Moving from the data plane to the control plane potentially enables more powerful analysis, as the former is generated by the latter. However, prior work is limited due to supporting only a single routing protocol (e.g., BGP in Bagpipe [41] and rcc [18]) or a limited set of routing protocol features (e.g., ARC [21]). Batfish [19] can reason about the entire control plane but its analysis is expensive because it simulates the individual steps of each routing protocol. In contrast, ERA enables fast exploration using a succinct encoding of control plane behavior.

Clean-slate control plane design: Metarouting [24], glue logic [33], and Propane [16] aim to build a correct-by-design control plane. While worthwhile in the long term, these efforts cannot reason about existing networks.

To summarize, what is critically missing today is the ability to efficiently explore the control plane involving various routing protocols. We illustrate this need below.

3 Motivation: Reachability Bugs

We motivate reasoning about multiple network incarnations using real reachability bugs encountered in a large cloud provider's network. These bugs were latent and manifested only under certain environments.

Maintenance-triggered: Some bugs stem from unexpected interactions of different routing protocols and configuration directives. In this example (Figure 2), the interactions are between static routing and BGP. For redundancy, the operator's goal was to have two paths between the DCN (datacenter network) and the WAN (wide area network), one through R_1 and the other through R_2. One day, the operator decided to temporarily bring down R_2 for maintenance, which she thought was safe because of the assumed redundancy. However, as soon as R_2 was brought down, the entire DCN was disconnected from the WAN (and the rest of the Internet).

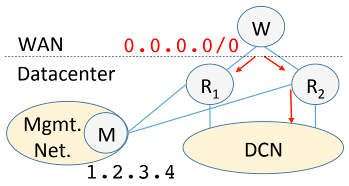


Figure 2: A bug triggered by maintenance.

Manual investigation revealed that R_1 contained a static default route `ip route 0.0.0.0/0 1.2.3.4` (here `1.2.3.4` is the next-hop of the static route, which is the address of the management network). Static routes to a prefix supersede dynamic routes [5, 8]. Thus R_1 preferred the static route over the default BGP route advertised by the WAN (shown in red). Since static routes are typically not propagated to neighbors, R_1 did not advertise the default route to the DCN. Thus, the DCN was entirely dependent on R_2 for external connectivity.

The bug in R_1's configuration was that the operator had forgotten to type keywords to indicate that the static route belonged to the management network, not data network. (These keywords were present in R_2's configuration.) The bug was latent as long as R_2 was up, but was triggered when R_2 was brought down.

Announcement-triggered: In Figure 3, DC_A had several services hosted inside the subprefixes of 10.10.0.0/16. Instead of announcing the individual subprefixes, R_1 was announcing this aggregate prefix. DC_B could reach the services inside DC_A through the WAN. As soon as a new service with prefix 10.10.1.160/28 was launched inside DC_A, all other services inside the /16 prefix became unreachable from DC_B.

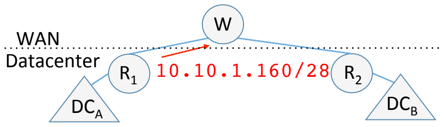


Figure 3: A bug triggered by a BGP announcement.

Investigation revealed two latent configuration bugs that combined to create this outage: *(1)* R_1 was not configured to filter 10.10.1.160/28 in its announcements to the WAN; and *(2)* R_2 was configured with an aggregate route to 10.10.0.0/16 with DC_B as the next hop. The result of the first bug was that the /28 announcement reached R_2 through the WAN. Then, as a result of the second bug, the /16 aggregate route was activated at R_2. This aggregate route, as a local route to router R_2, took precedence over the /16 being announced through the WAN. When the aggregate route was activated, R_2 started dropping all traffic to the /16 except for traffic to the /28. These drops are due to the *sinkhole* semantics of route aggregation—

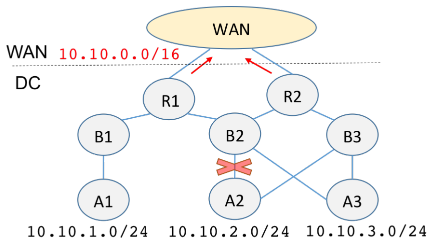


Figure 4: A bug triggered by link failure.

the aggregating router drops packets for subprefixes for which it does not have an active route to prevent routing loops [34].[1] Proper connectivity existed prior to the /28 announcement because the /16 announcement from the WAN did not activate the aggregate route at R_2.

Failure-triggered: In Figure 4, R_1 and R_2 were configured to announce prefix 10.10.0.0/16 that aggregated the subprefixes announced by leaf routers (A_1, A_2, A_3). After link A_2—B_2 failed, WAN traffic destined to A_2's prefix (10.10.2.0/24) started getting blackholed (i.e., dropped) at R_1 even though A_2 had connectivity via B_3 and R_2.

This blackhole was created because R_1 continued to make the aggregate announcement after the failure of link A_2–B_2, as it was still hearing announcements for the other two subprefixes corresponding to A_1 and A_3 (aggregate routes are announced as long as there is at least one subprefix present). As a result, the WAN sent (some) traffic for 10.10.2.0/24 toward R_1. But R_1 dropped those packets per the *sinkhole* semantics (see above).

4 ERA Overview

In this section, we present our approach and discuss the challenges in realizing it. Our target is a (datacenter, enterprise, or ISP) network of a realistic size (e.g., a few to hundreds of routers). As shown in Figure 5, our user is a network operator responsible for configuring routers. The operator has a set of intended reachability policies of the form "Router port A can talk to router port B" (as we will discuss in §7, several other practical policies are derivatives of "A can talk to B"). ERA allows operators to input their assumptions on what the network's environment will send (e.g., based on relationship with peers/providers). It then analyzes the network's behavior under these assumptions and checks whether the behavior satisfies the intended reachability policies. This process can then be iterated with other environmental assumptions, in order to cover a range of possible environments.

[1]For instance, if W announced the default route to R2, R2 would forward traffic for 10.10.2.2 to W, which may then forward them to R2 (because R2 announces the aggregate /16 to W), and so on.

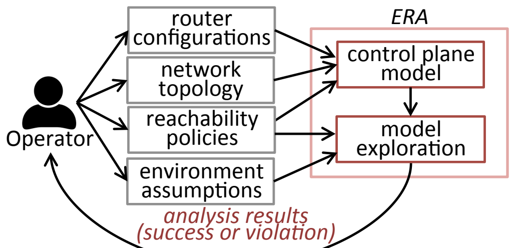


Figure 5: High-level vision of ERA.

4.1 Our Approach

Here we give the intuition behind our approach to control plane analysis.

Relationship between data and control planes: The data plane takes as input a packet on a router port and moves the (possibly modified) packet to another port (on the same or a neighboring router). Thus, we can think of the data plane as a function of the form $DP : (pkt, port) \rightarrow (pkt, port)$. The data plane itself is generated by the control plane function given routers' configuration files, the network topology (i.e., which router ports are inter-connected), and the current environment (which captures the route advertisements sent to the network by the "outside world") of the network: $CP : (env, Topo, Configs) \rightarrow DP(.)$.

Reachability policies via control plane analysis: Since packets are forwarded by the data plane, it is natural to think of an intended reachability policy $\phi_{A\rightarrow B}$ as a predicate that indicates whether a given packet should be able to reach from router port A to router port B. We say data plane DP is policy-compliant if $\phi_{A\rightarrow B}(pkt, DP)$ evaluates to $true$ for all A-to-B packets.

A seemingly natural approach for finding latent bugs is to produce the data plane associated with a given environment and then check reachability on that data plane [19]. However, this approach makes it prohibitively expensive to iteratively check multiple environments (§9.2). This is because for each possible environment (of which there are many), to compute the resulting data plane, we need to account for all low-level message passings and nuances of routing protocols. Instead, we want to be able to reason about the network directly at the level of the control plane and without explicitly computing the data plane.

To this end, our insight is as follows. Rather than producing the data plane that results from a given environment, we can analyze the control plane under that environment to determine i) the routes that each router in the network learns via its neighbors (e.g., a BGP advertisement) or its configuration file (e.g., static routes); and ii) the best route when multiple routes to the same prefix are learned. We can then use this information to directly check reachability.

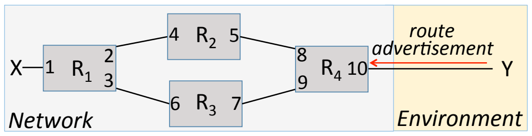


Figure 6: X-to-Y reachability depends on routers configurations and the environment.

An illustrative example: To visualize what it means to reason about reachability using control plane analysis, consider the example shown in Figure 6. Here we want to see what traffic reaches from port X to port Y so that we can check whether it is policy-compliant. From the figure we can see that to find the above traffic, we can try to find the routes that traverse the opposite direction on each of the two paths. Let $T^{i\rightarrow j}_{Router}(route)$ show the output of the configured router $Router$ on its port j given the input $route$ on its port i. (Intuitively, $route$ can be thought of as an abstraction for a route advertisement. The following section will elaborate on this abstraction.) If we knew $T(.)$, the answer would be:
$T^{2\rightarrow 1}_{R_1}(T^{5\rightarrow 4}_{R_2}(T^{10\rightarrow 8}_{R_4}(env))) \cup T^{3\rightarrow 1}_{R_1}(T^{7\rightarrow 6}_{R_3}(T^{10\rightarrow 9}_{R_4}(env)))$. The argument env here represents the assumptions that the user makes about the environment.

4.2 Challenges

Control plane-based reachability analysis requires us to address two key challenges:

- **An expressive and tractable control plane model:** To be expressive, this model needs to capture key behaviors of diverse protocols (e.g., BGP, OSPF route advertisements). A naive model (e.g., capturing protocol-specific behaviors verbatim), while expressive, is impractical because it will be too complex to explore. On the other extreme, a very high-level model (e.g., ignoring protocol-specific behaviors altogether) may be tractable to explore, but not expressive (e.g., BGP and OSPF have different ways of preferring routes).
- **Scalable control plane exploration:** Once we have a control plane model, we need the ability to efficiently explore the model with respect to the environment, in order to identify violations of intended reachability policies.

We tackle these challenges in §5 and §6, respectively.

4.3 Scope and Limitations

ERA's analysis requires the user to provide assumptions on the environment (or defaults to assuming that the environment makes all possible route announcements). If these assumptions are incorrect or overly permissive, then ERA can produce false positives, identifying purported errors that in fact will never arise in practice; e.g., a reputable ISP is not likely to hijack its peer's traffic. ERA is designed to have no other source of false positives (i.e., its control plane model is accurate). Though we have not formally proven this yet, empirically speaking, all the bugs that ERA has identified were real bugs.

ERA also has several sources of false negatives. First, ERA will only find bugs under environments specified as inputs and cannot guarantee the absence of bugs under all environments (unless exhaustively iterated on all possible environments). Second, certain classes of errors cannot be found by ERA by design. Specifically, ERA assumes that routing will converge and only analyzes this convergent state, which is key to efficient exploration of the control plane. Therefore convergence errors as well as reachability errors in transient states of the network will not be found (e.g., [23, 25]).

Finally, while ERA supports most of the common configuration directives, our current implementation does not support certain directives such as regular expressions in routing filters. Keeping up with configuration directives is a software engineering challenge due to their large and growing universe. Such limitations, however, are not fundamental to the design of ERA (unlike ARC [21], where the design itself cannot handle certain routing features).

As we will see in §9, ERA can find a large class of real-world bugs despite these limitations.

5 Modeling the Control Plane

We now describe our model for the network control plane. It $i)$ captures all routing protocols using a common abstraction; $ii)$ is expressive with respect to routing behaviors of individual protocols; and $iii)$ lends itself to scalable exploration. At a high level, we identify key behaviors of the control plane (e.g., route selection, route aggregation) and compactly encode them using binary decision diagrams (BDDs) [30].

Since the network control plane is a composition of the control planes of individual routers, we break down the problem of modeling the network control plane into modeling *(i)* the I/O unit of a router's control plane (§5.1), and *(ii)* the processing logic of a router's control plane (§5.2).

5.1 Route as the Model of Control Plane I/O

A naive way of modeling the I/O unit of the control plane of a router is to use the actual specification of route advertisements of different routing protocols, including their low-level details (e.g., keep-alive messages, sequence numbers [3, 9]). While expressive, such an I/O unit makes the control plane model too cumbersome. Conversely, if we completely ignore differences across protocols to simplify our I/O unit model, such a model may not be expressive; e.g., it cannot capture the fact that if a router learns two routes to the same destination prefix from two different routing protocols, the one offered by the protocol that has a smaller administrative distance (AD) will be selected [5, 8]. (We will see an example bug scenario due to this effect in §9.1.2, Figure 15b.)

Dst IP (32 bits)	Dst mask (5 bits)	Administrative distance (4 bits)	Protocol attributes (87 bits)

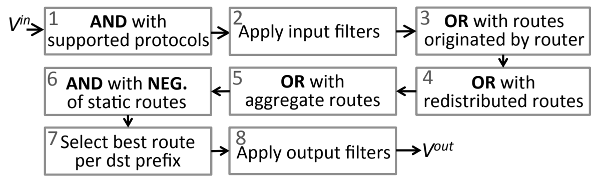
Figure 7: *route* as the model of control plane I/O.

To strike a balance between expressiveness and tractability, we introduce the notion of an abstract *route* as a succinct yet expressive I/O unit for the control plane model. Conceptually, a route mimics a route advertisement. It is a succinct bit-vector conveying key information in route advertisements that affects routing decisions of a router (see Figure 7). While not fundamental to our design, we have chosen a 128-bit vector to encode a route to enable fast CPU operations as we will discuss in §6.2. To accommodate diverse routing protocols, a route unifies key attributes of various protocols that affect a router's behaviors (i.e., administrative distance and protocol-specific route attributes).[2] To improve scalability, a route abstracts away the low-level nuances of actual protocols (e.g., seq. numbers, acknowledgements).

The fields of our route abstraction are:

- *Destination IP and mask*: Together, they represent the destination prefix that the route advertises. To make a route compact, we store the mask in 5 bits (instead of its naive storage in 32 bits). For completeness, Appendix A shows the details of how we do this.
- *Administrative distance (AD)*: This is a numerical representation of the routing protocol (e.g., BGP, OSPF) of the route such that $AD_A < AD_B$ denotes routing protocol A is preferred to protocol B.
- *Protocol attributes*: This captures protocol-specific attributes of the routing protocol represented by AD. For example, if the value of AD corresponds to BGP, the protocol attributes field encodes the BGP attributes (i.e., weight, local preference). To enable fast implementation of route selection in our router model (that we will discuss in §5.2), we carefully encode the attributes so that preferring a route between two routes $route_1$ and $route_2$ simply becomes a matter of choosing the smaller of two bit-vectors $AD_1.attrs_1$ and $AD_2.attrs_2$ when interpreted as unsigned integers (the symbol . denotes concatenation of the AD and protocol attributes fields of a route). For example, since route selection in BGP involves checking a prioritized list of BGP attributes (e.g., first checking the weight,

[2]Since our route model resembles routing messages in distance-vector protocols, we accommodate link state protocols (e.g., OSPF) by letting the attributes refer to the routes output by the Dijkstra algorithm.



V^{in}
1 AND with supported protocols
2 Apply input filters
3 OR with routes originated by router
4 OR with redistributed routes
5 OR with aggregate routes
6 AND with NEG. of static routes
7 Select best route per dst prefix
8 Apply output filters
V^{out}

Figure 8: High-level router model processing boolean representation of input routes.

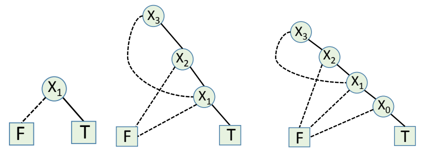
then local preference, etc.) [4], for a BGP route, the highest order bits of the protocol attributes field of the route encode the complement of the BGP weight attribute, followed by the complement of the local preference, and so forth. Note that the designated 87 bits for succinctly capturing protocol attributes have been sufficient in a range of realistic scenarios we have considered (§9), but there might be scenarios where more bits are needed to encode many distinct attributes.

5.2 Control Plane as a Visibility Function

Given the I/O unit of the control plane, next we need to model the processing logic that a router applies to input routes. Intuitively the router model is a function that given a route as its input, computes the corresponding output route(s). We identify 5 key operations of the router control plane: *(i) Input filtering*, which modifies/drops incoming route advertisements to the router; *(ii) Route redistribution*, which is necessary to capture cross-protocol interactions [31,33]; *(iii) Route aggregation*, which is a common mechanism to shrink forwarding tables, yet its improper use can lead to reachability violations [34]; *(iv) Route selection*, which is in charge of selecting the best route to a given destination prefix; and *(v) Output filtering*, which modifies/drops outgoing route advertisements.

Unfortunately, reasoning about the control plane one routing announcement at a time is not scalable. Instead, we *lift* our router model to work simultaneously on a *set* of route announcements. We refer to our router model as the *visibility function* because it captures how the router control plane processes the routing information made visible (i.e., given as input) to it. The input to the router visibility function, V^{in}, is the set of input routes sent by its neighbors and configured static routes; and its output, V^{out}, is the set of corresponding output routes that are sent downstream by the router. The notation $V^{out}_{Router} = T_{Router}(V^{in}_{Router})$ denotes the control plane visibility function of $Router$.

For fast exploration, we use BDDs to symbolically encode the set of I/O routes in a router model. A BDD is a compressed representation of a boolean function that enables fast implementation of operations such as conjunction, disjunction, and negation [30]. Our BDD encoding enables fast router operations by transforming operations on sets to quick operations on BDDs. For example, taking the complement of a set simply requires flipping the true/false leaves of the corresponding BDD.



(a) RIP. (b) Static route. (c) Output filter.

Figure 9: Example router model as a BDD. Dashed and solid lines represent the values 0 and 1 of the corresponding binary variable, respectively.

Figure 8 shows the high-level procedure for processing a boolean representation of sets of routes. (For completeness, the pseudocode for this is presented in Appendix B.) The steps to turn V^{in} into V^{out} are as follows:

1. *Supported protocols:* First, the routing protocols present in the configuration file are accounted for.
2. *Input filtering*: Then, the input filters are applied.
3. *Originated routes*: In addition to the input route, there are routes that directly stem from the configuration files, which are conceptually ORed with the input.
4. *Route redistribution*: A route redistribution command propagates routing information from a routing protocol (e.g., BGP) into another protocol (e.g., OSPF).
5. *Route aggregation*: If the router receives any input route that is more specific than any configured aggregate route, the aggregate route gets activated.
6. *Static routes*: A static route is a route locally known to the router (i.e., not shared with its neighbors). Further, by default, static routes take precedence over dynamic routes (e.g., OSPF, BGP, RIP, IS-IS) due to having a lower AD value. This behavior is captured by ANDing the negation of static routes with all other routes.
7. *Route selection*: Selecting the best of multiple routes to a destination prefix works as follows: *(i)* if the routes belong to different routing protocols, the one with the lowest AD value is selected, *(ii)* if the routes belong to the same routing protocol, the protocol-specific attributes determine the winner.
8. *Output filtering*: The router applies its output filters.

An illustrative example: We illustrate the procedure of Figure 8 using a small example. For ease of presentation, a route here has only 4 bits $x_3x_2x_1x_0$, with two bits x_3x_2 representing IP prefix, the bit x_1 representing AD, and the bit x_0 representing protocol attributes. A bar over a binary variable denotes its negation. In this example, the network operator assumes the router accepts *all* routes as input, which is captured by setting $V^{in} = 1$ (i.e., $true$).

Suppose a router is configured with a static route and RIP, with AD values of 0 and 1, respectively. Figure 9 shows the BDD representation of the router that has the following four (simplified) configuration commands:

- `RIP`, denoting the presence of RIP on the router, is captured by $1 \wedge x_1 = x_1$, as shown in Figure 9a.
- `static 10/2`: Since this static route overrides the RIP routes with the same prefix, the resulting predicate is $(\overline{x_3}\,\overline{x_2})x_1 = \overline{x_3}x_1 \vee x_2x_1$. This is shown in Figure 9b.
- `output filter: if RIP attribute is 0, make it 1`: The effect of the filter is to replace all occurrences of x_1 by x_1x_0. The resulting predicate is $\overline{x_3}x_1x_0 \vee x_2x_1x_0$. This is captured in Figure 9c.

Intuitively, the output $V^{out} = \overline{x_3}x_1x_0 \vee x_2x_1x_0$, simplified to $V^{out} = (\overline{x_3} \vee x_2) \wedge x_1x_0$, represents the fact that given $every$ environment as the input, the router outputs RIP (noted by x_1) with attribute 1 (noted by x_0) and the dest. prefix can be 00, 01, or 11 (noted by $\overline{x_3} \vee x_2$).

In the following section, we will discuss how to reason about the reachability behaviors of the network by exploring the router model we developed in this section.

6 Exploring the Model

Our reachability analysis is based on an exploration of the control plane model above. We first describe this exploration, and then describe how we leverage our BDD-based encoding to devise a set of scalable exploration mechanisms that use *(i)* the Karnaugh map, *(ii)* equivalence classes, and *(iii)* vectorized CPU instructions.

6.1 Exploration Method

We present our approach to finding traffic reachable from port A to port B using a representative example. Consider the scenario shown in Figure 10. The red path is an A-to-B path involving routers $R_A, \ldots, R_i, R_{i+1}, \ldots, R_B$. For ease of presentation, in this example, there is only one path from A to B; the general pseudocode presented in Appendix C accounts for all A-to-B paths.

To see the effect of the environment, consider router R_i, which has three paths to router ports that face the outside world (namely, outside facing ports of routers R_1, R_3, and R_5). Unless the operator makes a more specific assumption on an environment input (i.e., what route advertisements the outside world will send to the network), ERA starts analysis using the boolean value $true$ (represented by a BDD with only one leaf with the value $true$), which represents the fact that $every$ possible route are provided by the environment. On the other hand, if the operator is able to make a more scoped assumption about the environment (e.g., based on expected routes from a neighbor), the starting environment will reflect the assumption.

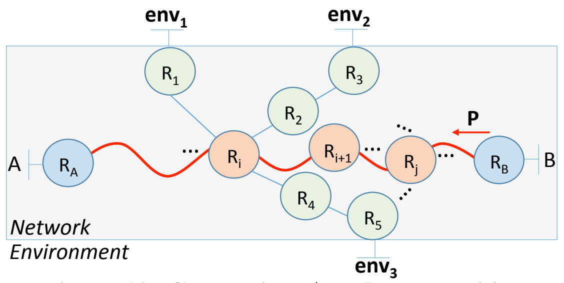


Figure 10: Computing A to B reachability.

Such assumptions can be encoded as a BDD that explicitly includes the relevant variables on the assumed prefix, administrative distance, or attributes values of incoming routes from the environment.

Computing traffic reachable from A to B involves the following steps:

1. *Applying the effect of the environment:* Every router on a A-to-B path that has a topology path to the environment, is affected by it. For router R_i in our examples, it means R_i receives the environment input E_i^{in}, where
$$E_i^{in} = T_1(env_1) \vee T_2(T_3(env_2)) \vee T_4(T_5(env_3))$$
2. *Computing routes reachable from B to A:* As we saw in §4.1, the key to computing traffic prefixes that reach from A to B using control plane analysis is to compute what route prefixes are made visible from B back to A. Let $assumed_B$ show the input the operator assumes about what port B receives from the environment. For the red path, this is captured by
$$reach_{A \rightsquigarrow B} = T_A(E_A^{in} \vee \ldots (T_{i+1}(E_{i+1}^{in} \vee \ldots T_N(E_B^{in} \vee assumed_B) \ldots)))$$
3. *Extracting prefixes reachable from A to B:* Since we are interested in route prefixes reachable from B to A, we eliminate binary variables in the route fields that do not correspond to prefix (i.e., AD and protocol attributes) in all boolean terms of $reach_{A \rightsquigarrow B}$.
4. *Accounting for on-path static routes:* In addition to the routes that reach from B to A, which cause traffic to reach from A to B, there is potentially other traffic that can reach from A to B due to static routes configured on on-path routers. This is because while a router does not advertise its static routes, activated static routes end up in its forwarding table. We account for such prefixes and OR them with the answer from step 3.
5. *Applying ACL rules affecting A-to-B traffic:* While a router configuration file primarily includes directives to configure the router control plane, it may include access control lists (ACLs) that restrict the actual traffic that can pass through the data plane of the router. We, therefore, account for ACLs by taking the result of step 4 and applying the ACLs of the on-path routers.

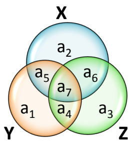


Figure 11: Visualization of predicates X, Y, and Z in terms of members of equivalence classes $a_1, \dots, a_7$.

Once traffic prefixes reachable from A to B are computed, the network is policy-compliant if the prefixes are equal to $\phi_{A \to B}$ from §4.1. If ϕ is violated, ERA applies the Karnaugh map [27] to the DNF representation of the violating routes to provide the human operator with fewer distinct items to investigate (§4.1); e.g., instead of reporting distinct prefixes $10.20.0.0/17$ and $10.20.128.0/17$ as violations, ERA summarizes and outputs them as $10.20.0.0/16$.

The process above finds policy violations in the context of a single set of environmental assumptions. The user can iterate multiple times with different assumptions in order to expose more errors. Conceptually, each iteration of ERA over a BDD input analyzes a *set* of concrete environments for which the network has an identical behavior. The analysis implicitly identifies this set during exploration, by accumulating constraints from the visibility function of each router in the network. Thus, the number of iterations needed for exhaustive exploration using ERA is far less than those needed with data plane based analysis tools such as Batfish.

6.2 Scalability Optimizations

To build an interactive tool for network operators, we want ERA to be able to compute $A - to - B$ reachability in no more than a few seconds. Even with the tractable control plane model that we developed in §5, a naive implementation of the exploration mechanism outlined in §6.1 fails to satisfy our goal. This is because of the very large range of possible environments. Here we present three techniques to scale control plane exploration.

Minimizing collection of routes with the K-map: As a first step, to minimize the binary representation of the router I/O, we apply the Karnaugh map (K-map), which is a common technique in circuit design [27].

Finding equivalence classes: Performing computations (e.g., conjunction and disjunction) on boolean representation of a real control plane is cumbersome. For example, the same or similar destination prefixes may appear on multiple routers. As such, if we encode prefixes naively, this may slow down control plane exploration.

Given this observation, before performing reachability analysis, ERA gets rid of redundant data by finding equivalence classes of routes which are treated identically by the network, using which the data can be rebuilt [42]. The advantage of doing so is that now performing disjunction and conjunction on boolean terms boils down to doing union and intersection on sets of integers (known as atomic predicates [42]). These integers are the indices of the equivalence classes. We illustrate this technique using an example. Suppose we need to compute the conjunction of the boolean terms X, Y, and Z (e.g., representing three routes). Instead of naively computing the conjunction on the raw boolean form of these terms, we do the following:

1. Express each term in terms of equivalence classes as depicted in Figure 11; e.g., $X = a_2 \lor a_5 \lor a_6 \lor a_7$.
2. Represent each term using the indices of members of equivalence classes, e.g., X is the union of members 2, 5, 6, and 7. (This way, irrespective of how bulky the raw form of term a_i might be, it is represented by integer value i.)
3. To compute $X \land Y \land Z$, intersect the sets of their corresponding indices: $\{1,5,6,7\} \cap \{1,4,5,7\} \cap \{3,4,6,7\} = \{7\}$, which indicates the answer to $X \land Y \land Z$ is a_7.

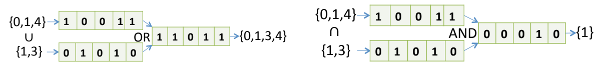
Implementing fast set operations: As we saw above, using equivalence classes, reachability analysis involves computing union and intersection of sets of integers. We leverage vectorized instructions on recent processors to perform fast set union and intersection of two sets of integers (i.e., the indices of the equivalence classes). The intuition is simple: if a set of integers is represented as a bit vector where each bit represents the presence/absence of the corresponding value, then the union (intersection) of two sets of integers is the bit-wise OR (AND) of the two bit vectors.

Figure 12 shows this approach using an example. In our implementation, we use instructions on 256-bit vectors in our Intel AVX2 implementation [13].

7 Going beyond Reachability

Building on basic A-to-B reachability, ERA can be used to check a wider range of policies. In §9, we will discuss scenarios involving these policies.

Valley-free routing: Operators often want to implement "valley-free" routing [20], which means that traffic from a neighboring peer or provider must not reach another such neighbor. This condition is a form of reachability policy that ERA can easily check.



(a) Set union using OR. (b) Set intersection using AND.

Figure 12: Fast ∪ and ∩ of two sets of integers.

Equivalence of two routers: Operators often use multiple routers to provide identical connectivity for fault tolerance. Checking if they are identically configured (e.g., using configuration syntax) is hard because the routers may be from different vendors and many aspects of the configuration (e.g., interface IP addresses) can legitimately differ across routers of even the same vendor. To check semantic equivalence of two routers' policies, we use the following property of BDDs: if two boolean functions defined over n boolean variables are equivalent (i.e., they generate the same output for the same input), their Reduced Ordered BDDs (ROBDDs) are identical [17]. In our implementation, we check the equality of the adjacency matrix representations of the BDDs of the two functions, which takes $O(n^2)$. In contrast, a brute force method will take $O(2^n)$.

Blackhole-freeness: A blackhole is a situation where a router unintentionally drops traffic. The blackholed traffic from A to B is the complement of the reachable traffic: $blackhole_{A \rightsquigarrow B} = \overline{reachability_{A \rightsquigarrow B}}$. Note that computing blackholes by ERA having computed reachability takes $O(1)$, as the negation of a BDD is the same BDD with its two leaves (corresponding to true and false) flipped.

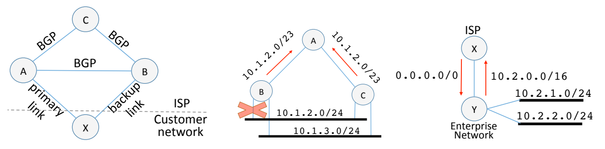
Waypointing: Operators may want traffic from A to B to go through an intended sequence of routers (e.g., to enforce advanced service chaining policies [15,35]). ERA checks waypointing by explicitly checking whether traffic reachable from A to B goes through the intended routers.

Loop-freeness: ERA can find permanent forwarding loops (e.g., created by static or aggregate routes—see Figure13c in §9.1) by checking whether the same router port appears twice in the reachability result.

8 Implementation

Our implementation of ERA [1] supports several configuration languages (e.g., Cisco IOS, JunOS, Arista). It uses Batfish's configuration parser, which normalizes a vendor-specific configurations to vendor-agnostic format. ERA, then, uses this vendor-agnostic format as input. We implement the control plane model, the K-map, and atomic predicates in `Java`. To operate on BDDs, we use the `JDD` library [7]. We implement our fast set intersection and union algorithms in `C` using Intel AVX2, which expands traditional integer instructions to 256 bits [13].

A natural question might be how much effort it takes to add support for various routing protocols to ERA. In our experience, this effort is minimal. It took two of the authors a few hours to model the common routing protocols because of two reasons. First, there are fewer than 10 common routing protocols (e.g., BGP, OSPF, RIP, IS-IS). Second, for each protocol, the key insight for creating the



(a) Violation of way-pointing [32]. (b) Black-hole [34]. (c) Permanent loop [34].

Figure 13: Finding known bugs in synthetic scenarios.

model is to know how the protocol prefers a route over another in the steady state, which is concisely defined in protocol specifications.

9 Evaluation

In this section, we evaluate ERA and find that:

- It can help find both known and new reachability violations (§9.1).
- It can scale to large networks (e.g., it can analyze a network with over 1,600 routes in 6 seconds), and our design choices are key to its scalability (§9.2);

9.1 Finding Reachability Bugs with ERA

We show the utility of ERA in finding reachability violations in scenarios involving known bugs as well as new bugs across both real and synthetic scenarios. These scenarios illustrate violations that are latent and get triggered only in certain environments (i.e., a certain router advertisement sent to the network by the routers located int the outside world). Even for scenarios involving only a small number of routers, existing network verification techniques lack the ability to find latent bugs (§2), and trying to extend these tools to enumerate different environments poses a serious scalability challenge (e.g., we will quantify this for Batfish, a recent network verification tool, in §9.2). Further, as we will discuss in §9.2, ERA scales to large networks (e.g., over 1,600 routers).

All experiments below were done under the assumption that the environment sends *all* possible route announcements, i.e. the BDD of each environmental input is simply the predicate *true*. Though this environmental assumption is not guaranteed to cover all possible environments, in practice it is effective at rooting out latent bugs due to its "maximal" nature, as we show below. This points out an important advantage of ERA over Batfish [19]. While both tools require an environment as input, Batfish's low-level simulation of routing protocols makes it prohibitively expensive to run with such a maximal environment, so in practice Batfish users must craft specific environments that are suspected to cause problems.

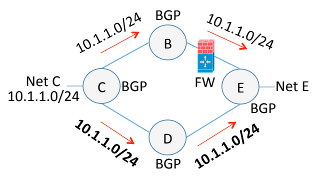
9.1.1 Finding Known Bugs in Synthetic Scenarios

- **Violation of waypointing due to route redistribution:** In this scenario borrowed from [32] and shown in Figure 13a, the customer wants to waypoint its traffic through $X - A - C$ and use $X - B - C$ as the backup path. However, static routes configured on routers A and B are redistributed into BGP, and the ISP advertises them into the rest of the Internet. As a result, $B - X$ acts as a primary link. (One way to prevent this would be for the customer to adjust the default AD values of BGP and static routes on B.)

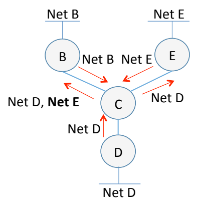
- **Blackhole due to route aggregation:** In this scenario borrowed from [34] and shown in Figure 13b, both routers B and C are configured to announce aggregate route 10.1.2.0/23 to router A. After the marked interface of B fails, B continues to announce the aggregate route, which causes A to send packets destined to 10.1.2.0/24 to B. B will drop this traffic, as the its link to the 10.1.2.0/24 subnetwork is down.
- **Permanent loop due to route aggregation:** In this scenario borrowed from [34] and shown in Figure 13c, the ISP router X advertises the default route 0.0.0.0/0 to router Y. Even though Y has connectivity to only 10.2.1.0/24 and 10.2.2.0/24, it has been configured to advertise to the ISP the aggregate route for the entire 10.2.0.0/16 prefix. Now since 10.3.0.0/24 is as subprefix of 10.2.0.0/16, the ISP may send traffic to destination prefix 10.3.0.0/24 to Y. Consequently, since Y does not know how to reach 10.3.0.0/24, this traffic will match its default route entry and be bounced back to the ISP. This traffic, therefore, will trap in a permanent loop between X and Y.

To further evaluate the effectiveness of ERA, we did a red team-blue team exercise. In each scenario, the red team introduced misconfigurations that cause a reachability violation unbeknownst to the blue team. Then the blue team uses ERA to check whether the intended policy is violated. Across all scenarios, the blue team successfully found the violation. Here is a summary of the scenarios:

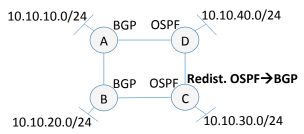
- **Violation of waypointing:** In Figure 14a, the intended policy is to ensure traffic originating from network E destined to network C goes through path $E - B - C$ (so that it is scrubbed by the firewall). However, this policy is violated because router E receives the prefix of network C from both routers B and D, which means $NetE \rightarrow NetC$ traffic may go through path $E - D - C$ skipping the firewall. The root cause of the problem was the fact that none of routers C, D, or E filtered the route advertisement for the 10.1.1.0/24 prefix on the $E - D - C$ path.
- **Violation of valley-free routing:** In Figure 14b, B and E are providers for C, which in turn, is a provider for D. A missing export filter on C caused C to advertise the prefix for $NetE$ to B. This is a violation of the valley-free routing property, specifically, due to customer C providing connectivity between two of its providers, namely, B and E.



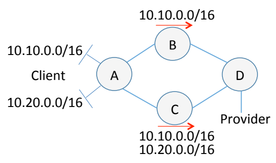
(a) Violation of waypointing via B.



(b) Not valley-free.



(c) Violation of isolation between $\{A, B\}$ and $\{C, D\}$.



(d) Misconfigured backup path $D - B - A$.

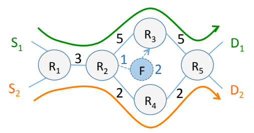
Figure 14: Finding known bugs in synthetic scenarios using the red-blue teams exercise.

- **Violation of intended isolation:** In Figure 14c, we want the traffic from segments $\{A, B\}$ (running BGP) and $\{C, D\}$ (running OSPF) to remain isolated from each other. However, this policy is violated due to a misconfiguration on C whereby OSPF is redistributed into BGP, that will allow traffic from $\{A, B\}$ to reach $\{C, D\}$.
- **Misconfigured backup path:** In Figure 14d, the client has two /16 networks connected to A and intends to maintain two paths to the provider to ensure reachability in case of failure on one of them. This policy is violated because of an incorrect filter configured on B that drops the advertisement for the 10.20.0.0/16 network. As a result, if path $D - C - A$ fails, the 10.20.0.0/16 network will be unreachable from the provider.

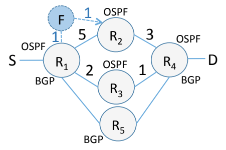
9.1.2 Finding New Bugs in Synthetic Scenarios

Finding reachability bugs in hybrid networks: Operators may prefer to opt for a hybrid network, which involves deploying SDN alongside traditional network routing infrastructure for scalability and fault tolerance [40]. Next we show how ERA can find policy violations arising in such hybrid deployments.

Fibbing [40] is a recent method to allow an operator to use an SDN controller to flexibly enforce way-pointing policies in a network running vanilla OSPF. The key primitive is “fibbing” whereby the SDN controller pretends to be a neighboring router and makes fake route advertisements with carefully crafted costs. For example, consider the network of Figure 15a, where links are annotated with their OSPF weights. If we run OSPF, both



(a) Route aggregation on R_2.



(b) Cross-protocol effects.

Figure 15: New bugs in a synthetic scenario involving hybrid (i.e., SDN-traditional) networks.

source to destination flows will take the cheaper path $R_1 - R_2 - R_4 - R_5$. Now, for load balancing purposes, the operator wants to make $S_1 \rightarrow D_1$ traffic take the path $R_1 - R_2 - R_3 - R_5$ without fiddling with OSPF wights. Fibbing will let her accomplish this by using a fake router F that claims to be able to reach D_1 at a cost of 2. As a result, now R_2 will start sending traffic destined to D_1 through F, as the new cost 1+2=3 is better than the cost 2+2=4 of going through R_4.

A hybrid network is particularly error-prone due to intricate interactions between SDN and traditional protocols. To show the utility of ERA in reasoning about such networks, we describe two scenarios:

- **Interaction between fibbing and aggregate routes:** In Figure 15a, the goal is to use fibbing to enforce the waypointing denoted by green and orange paths. We used ERA to find a violation of this policy. The root cause was an aggregate route configured on R_2 to destination prefix $D_1 \cup D_2$ pointing to R_4 as its next hop. As a result, both $S_1 \rightarrow D_1$ and $S_2 \rightarrow D_2$ traversed the orange path, which violated the policy.
- **Cross-protocol effects:** In Figure 15b, the goal is to use fibbing to waypoint traffic to D through $R_1 - R_2 - R_4$. We used ERA to find a violation of this in a red team-blue team exercise. Each router in the figure is annotated with the routing protocol(s) it runs. Router R_4 had a static route to D that is redistributed into BGP and OSPF. As a result, router R_1 received route advertisements for D from both OSPF (from R_2 and R_3) and BGP (from R_5). Now since BGP, by default, has a lower AD value than OSPF, R_1 chose the advertisement offered by R_5! Therefore, fibbing here fails to enforce the waypointing policy.

Fibbing is proven to be correct [40], but only if the network merely runs OSPF. The takeaway from the above scenarios is that for hybrid networks to be practical, we need to account for realistic router configurations (e.g., route aggregation by R_2 in Figure 15a) and cross-protocol interactions (e.g., BGP/OSPF in Figure 15b).

Note that finding arbitrary SDN bugs is beyond the scope of ERA. ERA handles SDN only if its behavior can be abstracted in our control plane model, in a manner similar to what we do for conventional routing protocols.

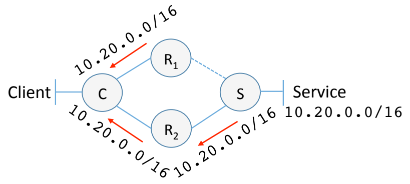


Figure 16: R_1 leaks the service prefix.

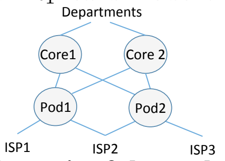


Figure 17: A schematic of the analyzed *CampusNet*.

9.1.3 Finding Known Bugs in Real Scenarios

Bugs reported in a cloud provider: The motivating scenarios we saw in §3 are based on bugs in a production network that we successfully reproduced using ERA.

Finding BGP route leaks: Roughly speaking, a route leak scenario involves: *(i)* a router incorrectly advertising the destination prefix of a service, and *(ii)* another router incorrectly accepting it. The combination of these results in absorbing traffic destined to the service on the wrong path, which can cause high-impact disruptions. Route leak is not a new problem (e.g., see AS 7007 incident [2]), but continues to plague the Internet to date (e.g., Google [6] and Amazon AWS [11] outages in 2015). To demonstrate the utility of ERA in proactive finding of route leak-prone configurations, we use a representative scenario shown in Figure 16. The intended path from the client to the service is through R_2; however, the client's traffic ends up taking the wrong path $C \rightarrow R_1$ because *(i)* R_1 incorrectly advertises the service prefix, and *(ii)* C prefers the route advertisement made by R_1 over the one made by R_2. ERA can proactively find route leaks, as a route leak is essentially a violation of waypointing. In this example, the traffic from client to server needs to be exclusively waypointed through R_2. We have synthesized a few route leak scenarios and used ERA to successfully find violations.

9.1.4 Finding New Bugs in Real Scenarios

Next we show the utility of ERA in finding new bugs in a campus (*CampusNet*) and a large cloud (*CloudNet*).

Finding new bugs in *CampusNet*: Figure 17 shows a simplified topology of the core of a large campus network, with a global footprint and over 10K users. The two core routers are in charge of interconnecting the three ISPs and the departments. There are two intended policies involving these four routers, both of which are violated:

- **Equivalence of core routers:** $Core2$ is meant to be $Core1$'s backup. ERA revealed that $Core1$ has OSPF configured on one of its interfaces, which is missing on $Core2$. As a result, if $Core1$ fails, the departments that rely on OSPF will be disconnected from the Internet.

- **Equivalence of pod routers:** $Pod1$ and $Pod2$, connecting the campus to the Internet, are both connected to $ISP2$ with the intention that link $Pod1 - ISP2$ is active and $Pod2 - ISP2$ is its backup. ERA revealed that the ACLs on $Pod1$ and $Pod2$ affecting their respective links with $ISP2$ are different. Specifically, $Pod2$ has more restrictive ACLs than $Pod1$. This means if link $Pod1 - ISP2$ fails, a subset of campus-to-$ISP2$ traffic will be mistakenly dropped by $Pod2$.

Finding new bugs in *CloudNet*: We used ERA to check equivalence of same-tier routers (analogous to routers R_1 and R_2 in Figure 2) on configurations of seven production datacenters of a large cloud provider. ERA revealed that seven routers in two datacenters had a total of 19 static routes responsible for violations of equivalence policies. The operators later removed all of these violating routes.

9.2 Scalability of ERA

Testbed: We run our scalability evaluation experiments on a desktop machine (4-core 3.50GHz, 16GB RAM).

Why not existing tools? The closest tool to ERA is Batfish [19], which *(1)* takes a concrete network environment; *(2)* runs a high-fidelity model of the control plane (e.g., low-level models of various routing protocols) to generate the data plane (i.e., routers forwarding tables); and *(3)* performs data plane reachability analysis. To put this in perspective, in an example scenario involving a chain topology with two routers, Batfish took about 4 seconds. In contrast, ERA took 0.17 seconds to analyze the same network (a 23X speedup over Batfish). Further, as mentioned earlier, Batfish's performance will degrade as the size of the environment increases, while ERA's BDD-based approach allows it to naturally handle even the "maximal" environment, represented by the BDD *true*.

Effect of optimizations: Table 1 shows the effect of our optimizations from §6.2, namely, the K-map, equivalence classes (EC), and fast set operations compared to a baseline involving use of BDDs without these optimizations. The tables shows the average values from 100 runs, each involving A-to-B reachability analysis between two randomly selected ports. Stanford [12] and Purdue [10] are campus networks, OTEGlobe [14] is an ISP, and FatTree is a synthetic datacenter topology. The takeaway here is that our optimizations yield a speedup of 2.5× to 17×, making ERA sufficiently fast to be interactively usable.

To see the effect of the type of policy on the analysis latency, we measured the analysis latency for all properties from §7 on the Purdue and OTEGlobe topology, none of which took more than 6.1 seconds. This is expected, as these policies are derivatives of reachability analysis.

Topo.	#routers/ave path len.	Reachability analysis latency (sec)			
		baseline	kmap	kmap+EC	ERA
Stanford	16/2	5	1.8	0.30	0.29
OTEglb	92/3.3	7.8	3.5	1.97	1.84
FatTree	1,024/5.89	13.8	7.01	6.1	5.4
Purdue	1,646/6.8	15	8	6.5	6

Table 1: Effect of our optimizations.

10 Conclusions

Since networks are constantly changing (e.g., new route advertisements, link failures), operators want the ability to reason about reachability policies across many possible changes. In contrast to prior work, which either focuses on a subset of the network's control plane or focuses on one incarnation of the network as represented by a single data plane, ERA models the entire control plane and checks network reachability directly in that model. Our design addresses key expressiveness and scalability challenges via a unified protocol-invariant routing abstraction, a compact binary decision diagram based encoding of the routers' control plane, and a scalable application of boolean operations (e.g., vector arithmetic).

We showed that ERA provides near-real-time analysis capabilities that can scale to datacenter and enterprise networks with hundreds of nodes and uncover a range of latent reachability bugs. While ERA does not automatically reason about all possible of environments, it helps find latent reachability bugs by allowing the users to specify a rich *set* environments using BDDs and quickly analyzing each such set. For instance, a particularly challenging environment, of all possible routing announcements from a neighbor, can be captured simply using BDD *true*.

In future work, we will identify conditions under which a single run of ERA is guaranteed to cover all possible environments and extend ERA to automatically explore all possible environments. Another natural direction for future work is to prioritize bug fixing based on the likelihood of occurrence and severity of aftermath, and to bring the human operator into the debugging and repair loop.

Acknowledgments

We thank our shepherd George Porter and the OSDI reviewers for their constructive feedback. This work was supported in part by NSF Awards CNS-1552481 and CNS-1161595 and by a VMware Graduate Fellowship.

Appendix

The goal of the following appendices is to provide details of our control plane modeling and exploration approach that we presented in §5 and §6.

A Computing Destination Prefix

To make our route abstraction compact, we store the destination mask field in 5 bits (instead naively storing it in 32 bits) as we saw in §5.1. Here we concretely describe how we do this. Let $dstIP$ and $dstMask$ denote a 32-bit destination IP address and our 5-bit encoding of the destination mask. To compute the destination prefix that the destination IP and mask represent, we first transform the mask to its customary 32-bit representation (e.g., 255.255.0.0), and then AND it with the IP address:

$dstPrefix \leftarrow dstIP\&((2^{32}-1) << (32-dstMask))$

where $<<$denotes the shift left operator.

B Route Visibility Function

For completeness, the pseudocode of Figure 18 shows the details of the router control plane model from §5.2. The pseudocode describes how a configured router turns the boolean representation of its input routes to output routes. Note that we account for per-port (i.e., router interface) behaviors because, in general, a router can have distinct routing behaviors configured on its different ports.)

1. The input to the router is the disjunctive normal form (DNF) boolean representation of input routes. This represents the input route(s) the environment of the router (i.e., its neighbors) sends to it (line 1). The output of the pseudocode is the DNF boolean representation of the output routes (line 2).
2. First, the routing protocols present in the configuration file are accounted for (lines 6-7).
3. Then, the input filters are applied to the input route (Lines 8-12).
4. In addition to the input route, there are route advertisements that directly stem from the configuration files (e.g., the `network` bgp configuration command). These are unioned with the input route in lines 13-14.
5. A route redistribution command propagates routing information from a routing protocol, denoted by *fromProto* (e.g., BGP) into another, denoted by *toProto*, (e.g., OSPF). This is captured in lines 15-22.
6. A route aggregation (a.k.a. route summarization) command works as follows: if the router receives any input route that is more specific than an aggregate route, the aggregate route is activated (lines 23-28).[3]

[3]In line 21 (line 27) of Figure 18, if there are explicit attributes configured for route redistribution (route aggregation), we use those values instead of default attributes.

```
1  ▷ Inputs: (1) Configuration information pertaining to router output port
     $Router_{port}$ including: static routes $sr[.]$, route redistribution $rr[.]$, route
     aggregation $ra[.]$, supported routing protocols $proto[.]$, input filters $if[.]$,
     output filters $of[.]$
     (2) Input to the router is a boolean function in DNF form:
     $V^{in} = X_1^{in} \vee \cdots \vee X_N^{in}$
2  ▷ Output: Boolean representation of $Router_{port}$ in DNF

3  ▷ Route bit vector from Figure 7, denoted by $X$, is concatenation of 3 fields:
     $X = X_{prefix}.X_{proto}.X_{attr}$
4  ▷ We show the length of an array $array$ by $size(array[.])$
5  $V^{out} = V^{in}$ ▷ Initializing the output

6  ▷ Applying supported routing protocols
7  $V^{out} = V^{out} \wedge \{\bigvee_i X_{proto[i]}\}$

8  ▷ Applying input filters
9  for $i = 1$ to $size(if[.])$
10   for each disjunctive term of $V^{out}$, denoted by $V_j^{out}$
11     if $V_j^{out}$ matches $if[i].condition$
12       apply action $if[i].action$

13 ▷ Accounting for routes that $Router$ originates, denoted by $V^{local}$
14 $V^{out} = V^{out} \vee V^{local}$

15 ▷ Applying route redistribution
16 for $i = 1$ to $size(rr[.])$
17   for each disjunctive term of $V^{out}$, denoted by $V_j^{out}$
18     if $V_j^{out}.X_{proto} == rr[i].fromProto$
19       $newTerm = V_j^{out}$
20       $newTerm.X_{proto} = rr[i].toProto$
21       $newTerm.X_{attr} = defaultAttr[proto]$
22       $V^{out} = V^{out} \vee newTerm$

23 ▷ Applying route aggregation
24 for $i = 1$ to $size(ra[.])$
25   $newTerm.X_{prefix} = ra[i].prefix$
26   $newTerm.X_{proto} = ra[i].proto$
27   $newTerm.X_{attr} = defaultAttr[proto]$
28   $V^{out} = V^{out} \vee newTerm$

29 ▷ Applying static routes
30 for $i = 1$ to $size(sr[.])$
31   for each disjunctive term of $V^{out}$, denoted by $V_j^{out}$
32     if $AD(V_j^{out}.X_{proto}) > AD(static)$
33       $V_j^{out} = V_j^{out} \wedge (\overline{sr[i].prefix})$

34 ▷ Applying route selection
35 for each prefix $prfx$ present in $V^{out}$
36   $precedence = +\infty$
37   for each disjunctive term of $V^{out}$, denoted by $V_j^{out}$
38     if $(V_j^{out}.prefix == prfx)\&\&(V_j^{out}.AD.attr < precedence)$
39       $precedence = V_j^{out}.AD.attr$ ▷ Finding best route
40   for each disjunctive term of $V^{out}$, denoted by $V_j^{out}$
41     if $(V_j^{out}.prefix == prfx)\&\&(V_j^{out}.AD.attr > precedence)$
42       Eliminate $V_j^{out}$ from $V^{out}$ ▷ Eliminating others
43   $V_j^{out} = V_j^{out} \vee prfx.precedence$

44 ▷ Applying output filters
45 for $i = 1$ to $size(of[.])$
46   for each disjunctive term of $V^{out}$, denoted by $V_j^{out}$
47     if $V_j^{out}$ matches $of[i].condition$
48       apply action $of[i].action$

49 return $V^{out}$
```

Figure 18: Route control plane visibility function.

7. A static route, if present in the configuration file, is a route locally known to the router (i.e., not shared with its neighbors). Further, by default, static routes have a

```
1   ▷ Inputs: (1) router-level topology of network
              (2) Set of router ports facing environment $Env$
              (3) routers configurations
2   ▷ Output: Prefix(es) of traffic reaching from router port $A$ to router
    port $B$

3   Parse router configurations into boolean functions (using Figure 18)
4   Initialize $assumed_e$ on port $e$ (by default, $true$)
5   initialize $assumed_B$ on port B (by default, $true$)

6   ▷ Accounting for effect of environment on routers on an $A$-to-$B$
    path
7   for each router $router_i$ on an $A - to - B$ path
8     for each environment-facing port $e \in Env$
9       for each path $p$ from port $e$ to $router_i$
10        ▷ $router_j$ is the $j$th router on $e \rightsquigarrow i$,
    where $1 \leq j \leq M(j)$
11        $E^{in}_{e\rightarrow i,p} = E^{in}_{e\rightarrow i,p} \vee T_{M(j)}(\dots(T_1(assumed_e))\dots)$
12      $E^{in}_{e\rightarrow i} = E^{in}_{e\rightarrow i} \vee V^{in}_{e\rightarrow i,p}$
13    $E^{in}_{i} = E^{in}_{i} \vee E^{in}_{e\rightarrow i}$

14  ▷ Compute per-path reachability
15  Find all paths from $B$ to $A$ in $G$:
    $Path_{B\rightsquigarrow A} = \{path^1_{B\rightsquigarrow A}, \dots, path^N_{B\rightsquigarrow A}\}$
16  ▷ $router^j_i$ is the $j$th router on $path^i_{B\rightsquigarrow A}$,
    where $1 \leq j \leq M(j)$
17  $reachability^{path^i_{B\rightsquigarrow A}}_{B\rightsquigarrow A} =$
    $T_{M(j)}(\dots(T_2(E^{in}_2 \vee (T_1(E^{in}_1 \vee assumed_B))))$

18  Eliminate binary variables in $reachability_{A\rightsquigarrow B}$ except those
    corresponding to $X_{prefix}$

19  ▷ Accounting for static routes
20  $static_{A\rightsquigarrow B} = \bigvee_i(\bigwedge_k(StaticPrefix_{Router^k_i}))$
21  $reachability_{A\rightsquigarrow B} = reachability_{A\rightsquigarrow B} \vee static_{A\rightsquigarrow B}$

22  ▷ Accounting for on-path ACLs. $Router^k_i$ is the $k$th router on
    $path^i_{A\rightsquigarrow B}$
23  $reachability^{path^i_{B\rightsquigarrow A}}_{B\rightsquigarrow A} =$
    $reachability^{path^i_{B\rightsquigarrow A}}_{B\rightsquigarrow A} \wedge (\bigvee_k ACLs_{Router^k_i})$

24  ▷ Compute all paths reachability
25  $reachability_{A\rightsquigarrow B} = \bigvee_i reachability^{path^i_{B\rightsquigarrow A}}_{A\rightsquigarrow B}$

26  return $reachability_{A\rightsquigarrow B}$
```

Figure 19: Computing A-to-B reachability.

lower AD value than dynamic routing protocols (e.g., OSPF, BGP, RIP, IS-IS), which makes them take precedence over these protocols. These behaviors are captures in lines 29-33.

8. Route selection is in charge of selecting the best route out of multiple routes to the same destination prefix: (i) if the routes belong to different routing protocols, the routing protocol with the lowest AD value is selected, (ii) if the routes belong to the same routing protocol, the protocol-specific attributes determine which one is selected. We have encoded protocol-specific attributes in such a way that a smaller value denotes a more preferred route. Route selection is shown in lines 34-43.
9. As lines 44-48 denote, the last operation of the router is applying the output filters.

C Computing Traffic Reachable from A to B

We saw the high-level procedure to compute the traffic reachable from port A to port B in the network in §6.1. For concreteness, here we present the pseudocode for doing so (Figure 19).

1. First, we account for the effect of the environment on the routers that are located on a path from A to B (lines 6-13).
2. The pseudocode then computes the routes that can reach from B to A over all paths between the two ports. For each path, we sequentially use the visibility functions of the on-path routers (lines 16-17). At this point, we have computed all route advertisement prefixes that reach from B to A, which is the traffic prefixes that reach from A to B.
3. Then, since we are interested in route prefixes reachable from B to A, we ignore route fields that do not correspond to prefix (i.e., AD and attributes) in line 18.
4. In addition to these prefixes, there is potentially other traffic that can reach from A to B to static routes configured on on-path routers. This is because while a router does not advertise its static routes, proper static routes end up in its forwarding table. By a proper static route we mean a static route that points to the next on-path router as its next hop. We account for static routes in lines 19-20.
5. Since routers ACL rules restrict what traffic prefixes will actually be forwarded, we then account for them in lines 22-23.

Finally, the computed per-path reachability results are unioned (lines 24-25).

References

[1] ERA. `https://github.com/Network-verification/ERA`.

[2] 7007 Explanation and Apology. `http://bit.ly/1e4djtW`.

[3] BGP Message Generation and Transport, and General Message Format. `http://bit.ly/1VMOI0R`.

[4] Border Gateway Protocol Path Selection. `http://bit.ly/1T1w7IH`.

[5] Cisco—What Is Administrative Distance? `http://bit.ly/1OkgevM`.

[6] Finding and Diagnosing BGP Route Leaks. `https://blog.thousandeyes.com/finding-and-diagnosing-bgp-route-leaks/`.

[7] JDD, a pure Java BDD and Z-BDD library. `https://bitbucket.org/vahidi/jdd/wiki/Home`.

[8] Juniper—Route Preferences. `http://juni.pr/1fQC4LY`.

[9] OSPF Message Formats. `http://bit.ly/1TvOwwL`.

[10] Purdue campus network configuration files. `https://engineering.purdue.edu/~isl/network-config/`.

[11] Route Leak Causes Amazon and AWS Outage. `https://blog.thousandeyes.com/route-leak-causes-amazon-and-aws-outage/`.

[12] Stanford campus network configuration files. `http://bit.ly/1rvoK5h`.

[13] The Intel Intrinsics Guide. `http://intel.ly/24sk3uz`.

[14] The Internet Topology Zoo. `http://www.topology-zoo.org/dataset.html`.

[15] High Performance Service Chaining for Advanced Software-Defined Networking (SDN) . `http://intel.ly/1ilX5PG`, 2014.

[16] R. Beckett, R. Mahajan, T. Millstein, J. Padhye, and D. Walker. Don't mind the gap: Bridging network-wide objectives and device-level configurations. In *Proc. SIGCOMM*, 2016.

[17] R. E. Bryant. Graph-based algorithms for boolean function manipulation. *IEEE Trans. Comput.*, 35(8):677–691, 1986.

[18] N. Feamster and H. Balakrishnan. Detecting BGP configuration faults with static analysis. In *Proc. NSDI*, 2005.

[19] A. Fogel, S. Fung, L. Pedrosa, M. Walraed-Sullivan, R. Govindan, R. Mahajan, and T. Millstein. A general approach to network configuration analysis. In *Proc. NSDI*, 2015.

[20] L. Gao. On inferring autonomous system relationships in the internet. *IEEE/ACM Trans. Netw.*, 9(6), Dec. 2001.

[21] A. Gember-Jacobson, R. Viswanathan, A. Akella, and R. Mahajan. Fast control plane analysis using an abstract representation. In *Proc. SIGCOMM*, 2016.

[22] A. Gember-Jacobson, W. Wu, X. Li, A. Akella, and R. Mahajan. Management plane analytics. In *Proc. IMC*, 2015.

[23] T. G. Griffin, F. B. Shepherd, and G. Wilfong. The stable paths problem and interdomain routing. *IEEE/ACM Trans. Netw.*, 10(2):232–243, Apr. 2002.

[24] T. G. Griffin and J. L. Sobrinho. Metarouting. In *Proc. SIGCOMM*, 2005.

[25] T. G. Griffin and G. Wilfong. An analysis of BGP convergence properties. In *Proc. SIGCOMM*, 1999.

[26] C. Guo, L. Yuan, D. Xiang, Y. Dang, R. Huang, D. Maltz, Z. Liu, V. Wang, B. Pang, H. Chen, Z.-W. Lin, and V. Kurien. Pingmesh: A large-scale system for data center network latency measurement and analysis. In *Proc. SIGCOMM*, 2016.

[27] F. J. Hill and G. R. Peterson. *Introduction to Switching Theory and Logical Design*. 1981.

[28] P. Kazemian, G. Varghese, and N. McKeown. Header space analysis: static checking for networks. In *Proc. NSDI*, 2012.

[29] A. Khurshid, W. Zhou, M. Caesar, and P. B. Godfrey. Veriflow: verifying network-wide invariants in real time. In *Proc. NSDI*, 2013.

[30] D. Knuth. *The Art of Computer Programming, Volume 4A: Combinatorial Algorithms, Part 1.* 2011.

[31] F. Le, G. G. Xie, D. Pei, J. Wang, and H. Zhang. Shedding light on the glue logic of the internet routing architecture. In *Proc. SIGCOMM*, 2008.

[32] F. Le, G. G. Xie, and H. Zhang. Instability free routing: beyond one protocol instance. In *Proc. CoNEXT*, 2008.

[33] F. Le, G. G. Xie, and H. Zhang. Theory and new primitives for safely connecting routing protocol instances. In *Proc. SIGCOMM*, 2010.

[34] F. Le, G. G. Xie, and H. Zhang. On route aggregation. In *Proc. CoNEXT*, 2011.

[35] W. Liu, H. Li, O. Huang, M. Boucadair, N. Leymann, Z. Cao, and J. Hu. Service Function Chaining (SFC) Use Cases. `http://bit.ly/1JTVneh`, 2014.

[36] N. P. Lopes, N. Bjørner, P. Godefroid, K. Jayaraman, and G. Varghese. Checking beliefs in dynamic networks. In *Proc. NSDI*, 2015.

[37] H. Mai, A. Khurshid, R. Agarwal, M. Caesar, P. B. Godfrey, and S. T. King. Debugging the data plane with Anteater. In *Proc. SIGCOMM*, 2011.

[38] D. A. Maltz, G. Xie, J. Zhan, H. Zhang, G. Hjálmtýsson, and A. Greenberg. Routing design in operational networks: A look from the inside. In *Proc. SIGCOMM*, 2004.

[39] G. Varghese. Technical perspective: Treating networks like programs. *Commun. ACM*, 58(11):112–112, Oct. 2015.

[40] S. Vissicchio, O. Tilmans, L. Vanbever, and J. Rexford. Central control over distributed routing. In *Proc. SIGCOMM*, 2015.

[41] K. Weitz, D. Woos, E. Torlak, M. D. Ernst, A. Krishnamurthy, and Z. Tatlock. Bagpipe: Verified BGP configuration checking. In *Proc. OOPSLA*, 2016.

[42] H. Yang and S. S. Lam. Real-time verification of network properties using atomic predicates. In *IEEE Transactions on Networking*, 2015.

[43] H. Zeng, P. Kazemian, G. Varghese, and N. McKeown. Automatic test packet generation. In *Proc. CoNEXT*, 2012.

Simplifying Datacenter Network Debugging with PathDump

Praveen Tammana
University of Edinburgh

Rachit Agarwal
Cornell University

Myungjin Lee
University of Edinburgh

Abstract

Datacenter networks continue to grow complex due to larger scales, higher speeds and higher link utilization. Existing tools to manage and debug these networks are even more complex, requiring *in-network techniques* like collecting per-packet per-switch logs, dynamic switch rule updates, periodically collecting data plane snapshots, packet mirroring, packet sampling, traffic replay, etc.

This paper calls for a radically different approach to network management and debugging: in contrast to implementing the functionality entirely in-network, we should carefully partition the debugging tasks between the edge devices and the network elements. We present the design, implementation and evaluation of PathDump, a minimalistic tool that utilizes resources at edge devices for network debugging. PathDump currently runs over a real network comprising only of commodity hardware, and yet, can support a surprisingly large class of network debugging problems. Evaluation results show that PathDump requires minimal switch and edge resources, while enabling network debugging at fine-grained time scales.

1 Introduction

Datacenter networks are essential to online services including web search, social media, online commerce, etc. Network outages and performance degradation, even if short-lived, can severely impact these services [1, 2, 3, 6]. Unsurprisingly, there has been a tremendous effort in building tools that allow network operators to efficiently manage networks and debug (the inevitable) network problems [17, 22, 24, 26, 30, 36, 39, 41].

As datacenter networks evolve to larger scales, higher speeds and higher link utilization, new classes of network problems emerge. Accordingly, over the years, network debugging tools have incorporated increasingly complex *in-network techniques* — capturing per-packet per-switch logs [17], collecting snapshots of entire data plane state [20, 21, 22, 26], dynamically updating switch rules [30], selective packet mirroring [33, 41], packet sampling [8, 16, 39, 41], active packet probes [9, 40, 41], traffic replay [37], a potpourri [41] — and this list barely scratches the surface of all the sophisticated techniques that have been proposed to be implemented on network switches for debugging purposes.

In this paper, we do not add to this impressive collection of techniques. Instead, we ask whether there are a non-trivial number of network debugging problems that obviate the need of sophisticated in-network techniques. To explore this question, we argue for a radically different approach: in contrast to implementing the debugging functionality *entirely* in-network, we should carefully partition the functionality between the edge devices and the network switches. Thus, our goal is *not* to beat existing tools, but to help them focus on a smaller set of nails (debugging problems) that we need a hammer (debugging techniques) for. The hope is that by focusing on a smaller set of problems, the already complex networks[1] and the tools for managing and debugging these networks can be kept as simple as possible.

We present PathDump, a network debugger that demonstrates our approach by enabling a large class of debugging problems with minimal in-network functionality. PathDump design is based on tracing packet trajectories and comprises of the following:

- Switches are simple; they neither require dynamic rule updates nor perform any packet sampling or mirroring. In addition to its usual operations, a switch checks for a condition before forwarding a packet; if the condition is met, the switch embeds its identifier into the packet header (*e.g.*, with VLAN tags).
- An edge device, upon receiving a packet, records the list of switch identifiers in the packet header on a local storage and query engine; a number of entries stored in the engine (used for debugging purposes) are also updated based on these switch identifiers.
- Entries at each edge device can be used to trigger and debug anomalous network behavior; a query server can also slice-and-dice entries across multiple edge devices in a distributed manner (*e.g.*, for debugging functionalities that require correlating entries across flows).

[1]as eloquently argued in [41]; in fact, our question about simpler network management and debugging tools was initially motivated by the arguments about network complexity in [41].

PathDump's design, by requiring minimal in-network functionality, presents several benefits as well as raises a number of interesting challenges. The benefits are rather straightforward. PathDump not only requires minimal functionality to be implemented at switches, but also uses minimal switch resources; thus, the limited switch resources [14, 28] can be utilized for exactly those tasks that necessitate an in-network implementation[2]. PathDump also preserves flow-level locality — information about all packets in the same flow is recorded and analyzed on the same end-host. Since PathDump requires little or no data transfer in addition to network traffic, it also alleviates the bandwidth overheads of several existing in-network debuggers [17, 33, 41].

PathDump resolves several challenges to achieve the above benefits. First challenge is regarding generality — what class of network problems can PathDump debug with minimal support from network switches? To get a relatively concrete answer in light of numerous possible network debugging problems, we examined all the problems discussed in several recent papers [17, 18, 30, 41] (see Table 2 in the appendix). Interestingly, we find that PathDump can already support more than 85% of these problems. For some problems, network support seems necessary; however, we show that PathDump can help "pinpoint" these problems to a small part of the network. We discuss the design, implementation and evaluation of PathDump for the supported functionality in §2.3 and §4.

PathDump also resolves the challenge of packets not reaching the edge devices (*e.g.*, due to packet drops or routing loops). A priori, it may seem obvious that PathDump must not be able to debug such problems without significant support from network switches. PathDump resolves the packet drop problem by exploiting the fact that datacenters typically perform load balancing (using ECMP or packet spraying [15]); specifically, we show that the difference between number of packets traversing along multiple paths allows identifying spurious packet drops. PathDump can in fact debug routing loops by leveraging the fact that commodity SDN switches recognize only two VLAN tags in hardware and processing more than two tags involves switch CPU (§4.5).

[2]As PathDump matures, we envision it to incorporate (potentially simpler than existing) in-network techniques for debugging problems that necessitate an in-network implementation. As network switches evolve to provide more powerful functionalities (*e.g.*, on-chip sampling) and/or larger resource pools, partitioning the debugging functionality between the edge devices and the network elements will still be useful to enable capturing network problems at per-packet granularity — a goal that is desirable and yet, infeasible to achieve using today's resources. Existing in-network tools that claim to achieve per-packet granularity (*e.g.*, Everflow [41]) have to resort to sampling to overcome scalability issues and thus, fail to achieve per-packet granularity.

Finally, PathDump carefully optimizes the use of data plane resources (*e.g.*, switch rules and packet header space) and end-host resources (*e.g.*, CPU and memory). PathDump extends our prior work, CherryPick [36], for per-packet trajectory tracing using minimal data plane resources. For end-host resources, we evaluate PathDump over a wide range of network debugging problems across a variety of network testbeds comprising of commodity network switches and end-hosts; our evaluation shows that PathDump requires minimal CPU and memory at end-hosts while enabling network debugging over fine-grained time scales.

Overall, this paper makes three main contributions:

- Make a case for partitioning the network debugging functionality between the edge devices and the network elements, with the goal of keeping network switches free from complex operations like per-packet log generation, dynamic rule updates, packet sampling, packet mirroring, etc.
- Design and implementation of PathDump[3], a network debugger that demonstrates that it is possible to support a large class of network management and debugging problems with minimal support from network switches.
- Evaluation of PathDump over operational network testbeds comprising of commodity network hardware demonstrating that PathDump can debug network events at fine-grained time-scales with minimal data plane and end-host resources.

2 PathDump Overview

We start with an overview of PathDump interface (§2.1), and PathDump design (§2.2). We then provide several examples on using PathDump interface for debugging network problems (§2.3, §2.4).

2.1 PathDump Interface

PathDump exposes a simple interface for network debugging; see Table 1. We assume that each switch and host has a unique ID. We use the following definitions:

- A `linkID` is a pair of adjacent `switchIDs` ($\langle S_i, S_j \rangle$);
- A `Path` is a list of `switchIDs` ($\langle S_i, S_j, \ldots \rangle$);
- A `flowID` is the usual 5-tuple (⟨srcIP, dstIP, srcPort, dstPort, protocol⟩);
- A `Flow` is a (⟨flowID, Path⟩) pair; this will be useful for cases when packets from the same `flowID` may traverse along multiple `Paths`.
- A `timeRange` is a pair of `timestamps` ($\langle t_i, t_j \rangle$);

[3]Available at: https://github.com/PathDump.

Host API	Description
`getFlows(linkID, timeRange)`	Return list of `flows` that traverse `linkID` during specified `timeRange`.
`getPaths(flowID, linkID, timeRange)`	Return list of `Paths` that include `linkID`, and are traversed by `flowID` during specified `timeRange`.
`getCount(Flow, timeRange)`	Return packet and byte counts of a `flow` within a specified `timeRange`.
`getDuration(Flow, timeRange)`	Return the duration of a `flow` within a specified `timeRange`.
`getPoorTCPFlows(Threshold)`	Return the `flowIDs` for which `protocol = TCP` and the number of consecutive packet retransmissions exceeds a threshold.
`Alarm(flowID, Reason, Paths)`	Raise an alarm regarding `flowID` with a reason code (e.g., TCP performance alert (`POOR_PERF`)), and corresponding list of `Paths`.
Controller API	**Description**
`execute(List⟨HostID⟩,Query)`	Execute a `Query` once at each host specified in list of `HostIDs`; a `Query` could be any of the ones from Host API.
`install(List⟨HostID⟩,Query,Period)`	Install a `Query` at each host specified in list of `HostIDs` to be executed at regular `Periods`. If the `Period` is not set, the query execution is triggered by a new event (*e.g.*, receiving a packet).
`uninstall(List⟨HostID⟩,Query)`	Uninstall a `Query` from each host specified in list of `HostIDs`

Table 1: PathDump Interface. See §2.1 for definitions and discussion.

PathDump supports wildcard entries for `switchIDs` and `timestamps`. For instance, ($\langle\star, S_j\rangle$) is interpreted as all incoming links for switch S_j and ($\langle t_i, \star\rangle$) is interpreted as "since time t_i".

Note that each host exposes the host API in Table 1 and returns results for "local" flows, that is, for flows that have this host as their `dstIP`. To collect the results distributed across PathDump instances at individual end-hosts, the controller may use the controller API — to `execute` a query, to `install` a query for periodic execution, or to `uninstall` a query.

2.2 PathDump Design Overview

The central idea in PathDump is to trace packet trajectories. To achieve this, each switch embeds its switchID in the packet header before forwarding the packet. However, naïvely embedding all the switchIDs along the packet trajectory requires large packet header space, especially when packets may traverse a non-shortest path (*e.g.*, due to failures along the shortest path) [36]. PathDump uses the link sampling idea from CherryPick [36] to trace packet trajectories using commodity switches. However, CherryPick supports commonly used datacenter network topologies (*e.g.*, FatTree, VL2, etc.) and does not work with arbitrary topologies. Note that this limitation on supported network topologies is merely an artifact of today's hardware — as networks evolve to support larger packet header space, PathDump will support more general topologies without any modification in its design and implementation.

An edge device, upon receiving a packet, extracts the list of switchIDs in the packet header and records them on a local storage and query engine (along with associated metadata, *e.g.*, flowID, timestamps, number of packets, number of bytes, etc.). Each edge device stores:

- A list of flow-level entries that are used for debugging purposes; these entries are updated upon each event (*e.g.*, receiving a packet).
- A static view of the datacenter network topology, including the statically assigned identifiers for each switch. This view provides PathDump with the "ground truth" about the network topology and packet paths.
- And, optionally, network configuration files specifying forwarding policies. These files are also used for monitoring and debugging purposes (*e.g.*, ensuring packet trajectories conform to specified forwarding policies). The operator may also push these configuration files to the end-hosts dynamically using the `Query` installation in controller API.

Finally, each edge device exposes the API in Table 1 for identifying, triggering and debugging anomalous network behavior. The entries stored in PathDump (within an edge device or across multiple edge devices) can be sliced-and-diced for implementing powerful debugging functionalities (*e.g.*, correlating entries across flows going to different edge devices). PathDump currently disregards packet headers after updating the entries to avoid latency and throughput bottlenecks in writing to persistent storage; extending PathDump to store and query at per-packet granularity remains an intriguing future direction.

2.3 Example applications

We now discuss several examples for network debugging applications using PathDump API.

Path conformance. Suppose the operator wants to check for policy violations on certain properties of the path taken by a particular `flowID` (*e.g.*, path length no more than 6, or packets must avoid `switchID`). Then, the controller may `install` the following query at the end-hosts:

```
Paths = getPaths(flowID, <*, *>, *)
for path in Paths:
  if len(path)>=6 or switchID in path:
    result.append (path)
if len(result) > 0:
  Alarm (flowID, PC_FAIL, result)
```

PathDump executes the query either upon each packet arrival, or periodically when a `Period` is specified in the query; an `Alarm()` is triggered upon each violation.

Load imbalance. Understanding why load balancing works poorly is of interest to operators because uneven traffic splits may cause severe congestion, thereby hurting throughput and latency performance. PathDump helps diagnose load imbalance problems, independent of the underlying scheme used for load balancing (*e.g.*, ECMP or packet spraying). The following example constructs flow size distribution for each of two egress ports (i.e., links) of interest on a particular switch:

```
result = {}; binsize = 10000
linkIDs = (l1, l2); tRange = (t1, t2)
for lID in linkIDs:
  flows = getFlows (lID, tRange)
  for flow in flows:
    (bytes, pkts) = getCount (flow, tRange)
    result[lID][bytes/binsize] += 1
return result
```

Through cross-comparison of the flow size distributions on the two egress ports, the operator can tell the degree of load imbalance. Even finer-grained diagnosis on load balancing is feasible; *e.g.*, when packet spraying is used, PathDump can identify whether or not the traffic of a flow in question is equally spread along various end-to-end paths. We demonstrate these use cases in §4.2.

Silent random packet drops. This network problem occurs when some faulty interface at switch drops packets at random without updating the discarded packet counters at respective interfaces. It is a critical network problem [41] and is often very challenging to localize.

PathDump allows a network operator to implement a localization algorithm such as MAX-COVERAGE [23]. The algorithm, as input, requires logs or observations on a network problem (that is, failure signatures). Using PathDump, a network operator can `install` a TCP performance monitoring query at the end-hosts for periodic monitoring (*e.g.*, period set to be 200 ms):

```
flowIDs = getPoorTCPFlows()
for flowID in flowIDs:
  Alarm (flowID, POOR_PERF, [])
```

Every time an alarm is triggered, the controller sends the respective end-host (by parsing flowID) the following query and collects failure signatures (that is, path(s) taken by the flow that suffers serious retransmissions):

```
flowID = (sIP, sPort, dIP, dPort, 6)
linkID = (*, *); tRange = (t1, *)
paths = getPaths (flowID, linkID, tRange)
return paths
```

The controller receives the query results (that is, paths that potentially include faulty links), locally stores them, and runs the MAX-COVERAGE algorithm implemented as only about 50 lines of Python code. This procedure repeats whenever a new alert comes up. As more path data of suffering TCP flows get accumulated, the algorithm localizes faulty links more accurately.

Traffic measurement. PathDump also allows to write queries for various measurements such as traffic matrix, heavy hitters, top-*k* flows, and so forth. The following query computes top-1000 flows at a given end-host:

```
h = []; linkID = (*, *); tRange = (t1, t2)
flows = getFlows (linkID, tRange)
for flow in flows:
  (bytes, pkts) = getCount (flow, tRange)
  if len(h) < 1000 or bytes > h[0][0]:
    if len(h) == 1000: heapq.heappop (h)
    heapq.heappush (h, (bytes, flow))
return h
```

To obtain top-*k* flows from multiple end-hosts, the controller can `execute` this query at the desired subset of the end-hosts.

2.4 Reducing debugging space

As discussed in §1, some network debugging problems necessitate an in-network implementation. One such problem is network switches incorrectly modifying the packet header — for some corner case scenarios, it seems hard for any end-host based system to be able to debug such problems.

One precise example in case of PathDump is switches inserting incorrect switchIDs in the packet header. In case of such network anomalies, PathDump may not be able to identify the problem. For instance, consider the path conformance application from §2.3 and suppose we want to ensure that packets do not traverse a switch s_1 (that

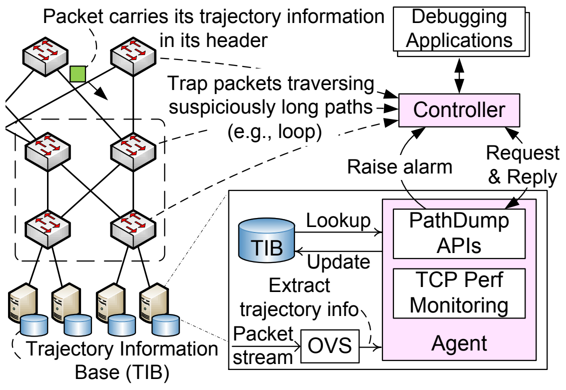


Figure 1: System overview.

is, `switchID=s₁` in the example). Suppose the packet trajectory $\{\texttt{src}, \texttt{s}_1, \texttt{s}_2, \ldots, \texttt{dst}\}$ actually involves $\texttt{s}_1$ and hence, PathDump must raise an alarm.

The main problem is that if $\texttt{s}_1$ inserts a wrong switchID, say $\texttt{s}_1'$, then PathDump will not raise an alarm. However, in many cases, the trajectory $\{\texttt{src}, \texttt{s}_1', \texttt{s}_2, \ldots, \texttt{dst}\}$ in itself will be infeasible — either because $\texttt{s}_1'$ is not one of the switchIDs or because the switch with ID $\texttt{s}_1'$ does not connect directly to either `src` or $\texttt{s}_2$. In such cases, PathDump will be able to trigger an alarm stating that one of the switches has inserted incorrect switchID; this is because PathDump continually compares the extracted packet trajectory to the ground truth (network topology) stored in PathDump.

3 PathDump Implementation

PathDump implementation comprises of three main components (Figure 1):

- In-network implementation for tracing packet trajectories using packet headers and static network switch rules (§3.1); PathDump's current implementation relies entirely on commodity OpenFlow features for packet trajectory tracing.
- A server stack that implements a storage and query engine for identifying, triggering and debugging anomalous network behavior (§3.2); we use C/C++ and Python for implementing the stack.
- A controller running network debugging applications in conjunction with the server stack (§3.3). The current controller implementation uses Flask [5] — a micro framework supporting a RESTful web service — for exchange of query-responses messages between the controller and the end-hosts.

We describe each of the individual components below. As mentioned earlier, PathDump implementation is available at `https://github.com/PathDump`.

3.1 Tracing packet trajectory

PathDump traces packet trajectories at per-packet granularity by embedding into the packet header the IDs of switches that a packet traverses. To achieve this, PathDump resolves two related challenges.

First, the packet header space is a scarce resource. The naïve approach of having each switch embed its switchID into the header before forwarding the packet would require large packet header space, especially when packets can traverse non-shortest paths (*e.g.*, due to failures along the shortest path). For instance, tracing a 8-hop path on a 48-ary FatTree topology would require 4 bytes worth of packet header space, which is not supported using commodity network components[4]. PathDump traces packet trajectories using close to optimal packet header space by using the link sampling idea presented in our preliminary work, CherryPick [36]. Intuitively, CherryPick builds upon the observation that most frequently used datacenter network topologies are very structured (*e.g.*, FatTree, VL2) and this structure enables reconstructing an end-to-end path by keeping track of a few carefully "sampled" links along any path. We provide more details below.

The second challenge that PathDump resolves is implementation of packet trajectory tracing using commodity off-the-shelf SDN switches. Specifically, PathDump uses the VLAN and the MPLS tags in packet headers along with carefully constructed network switch rules to trace packet trajectories. One key challenge in using VLAN tags is that the ASIC of SDN switch (*e.g.*, Pica8 P-3297) typically offers line rate processing of a packet carrying up to two VLAN tags (*i.e.*, QinQ). Hence, if a packet somehow carries three or more tags in its header, a switch attempting to match TCP/IP header fields of the packet would trigger a rule miss and usually forward it to the controller. This can hurt the flow performance. We show that PathDump can enable per-packet trajectory tracing for most frequently used datacenter network topologies (*e.g.*, FatTree and VL2), even for non-shortest paths (*e.g.*, up to 2 hops in addition to the shortest path), using just two VLAN tags. Note that these limitations on supported network topologies and path lengths are merely an artifact of today's hardware — PathDump achieves what is possible with today's networks, and as networks evolve to support larger packet header space, PathDump will support more general topologies (*e.g.*, Jupiter network [34]) and/or longer path lengths without any modification in its design and implementation.

[4]We believe networks will evolve to support larger packet header space. We discuss how PathDump could exploit this to provide even stronger functionality. However, we do note that even with availability to larger packet header space, ideas in PathDump may be useful since this additional packet header space will be shared by multiple applications.

However, not all non-shortest paths need to be saved and examined at end-hosts. In particular, when a path is *suspiciously long*, instant inspection at the controller is desirable while packets are on the fly; it may indeed turn out to be a serious problem such as routing loop. PathDump allows the network operator to define the number of hops that would constitute a suspiciously long path (we use 4 hops in addition to the shortest path length as default because packets rarely traverse such a long path in datacenter networks).

To keep the paper self-contained, we briefly review the ideas from CherryPick [36] below; we refer the readers to [36] for more detailed discussion and evaluation. We then close the subsection with a discussion on identifying and trapping packets traversing a suspiciously long path.

Tracing technique: CherryPick [36]. The need for techniques like CherryPick is clear; a naïve approach of embedding link ID of each hop into the packet header simply does not work [36]. Assuming 48-port switches, embedding a 6-hop path requires 36 bits in the header space whereas two VLAN tags only allow 24 bits.

The core idea of CherryPick is to sample links that suffice in representing an end-to-end path. One key challenge is that sampling links makes a local identifier inapplicable. Instead, each link should be assigned a global identifier. Clearly, the number of physical links is far more than that of available link IDs (c.f., 4,096 unique link IDs expressed in a 12 bit VLAN identifier vs. 55,296 physical links in a 48-ary fat-tree topology).

In addressing the issue, the following observation is used: aggregate switches between different lower level blocks (e.g., pods) must be interconnected only through core switches. Therefore, instead of assigning global IDs for the links in each pod, it becomes possible to share the same set of global IDs across pods. In addition, the scheme efficiently assigns IDs to core links by applying an edge-coloring technique [13]. The following describes how the links should be picked for fat-tree and VL2:

• *Fat-tree:* A key observation in it is that given any 4-hop path, when a packet reaches a core switch, the ToR-aggregate link it traversed becomes easily known, and there is only a single route to destination from the core switch. Hence, to build the end-to-end path, it is sufficient to pick one aggregate-core link that the packet traverses. When the packet is diverted from its original shortest path, the technique selects one extra link every additional 2 hops. Thus, two VLAN tags make it feasible to trace any 6-hop path. The mechanism is easily converted into OpenFlow rules (see [36]). The number of rules at switch grows linearly over switch port density.

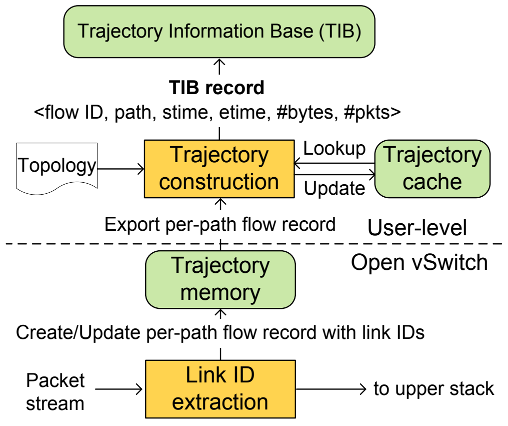


Figure 2: Trajectory information update procedure.

• *VL2:* VL2 requires to sample three links for tracing any 6-hop path. Hence, we additionally use DSCP field. However, because the field is only 6-bits long, we use it in order to sample an ToR-aggregate link in pod where there are only k links. After the DSCP field is spent, VLAN tags are being used over a subsequent path. If a packet travels over a 6-hop path, it carries one DSCP value and two VLAN tags at the end. In this way, rule misses on data plane is prevented for packets traversing a 6-hop path. We need two rules per ingress port: one for checking if DSCP field is unused, and the other to add VLAN tag otherwise, thus still keeping low switch rule overheads.

Given a 12-bit link ID space (*i.e.*, 4,096 link IDs), the scheme supports a fat-tree topology with 72-port switches (about 93K servers). Since DSCP field is additionally used for VL2, the scheme can support a VL2 topology with 62-port switches (roughly 19K servers).

Instant trap of suspiciously long path. PathDump by design supports identifying and trapping packets traversing a suspiciously long path. When a packet traverses one such path, it cannot help but carry at least three tags. An attempt to parse IP layer for forwarding at switch ASIC would cause a rule miss and the packet is sent to the controller. The controller then can immediately identify the suspiciously long path. We leverage this ability of PathDump to implement a real-time routing loop detection application (see §4.5).

3.2 Server stack

The modules in the server stack conduct three tasks mainly. The first is to extract and store the path information embedded in the packet header. Next, a query processing module receives queries from the controller, consumes the stored path data and provides responses. The final task is to do active monitoring of flows' performance and prompt raise of alerts to the controller.

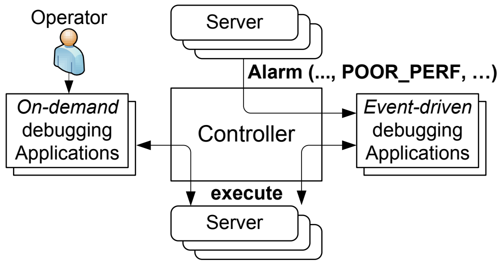
Trajectory information management. The trajectory information base (TIB) is a repository where packet trajectory information is stored. Because storing path information of individual packets can waste too much disk space, we do per-path aggregation given a flow. In other words, we maintain unique paths and their associated counts for each flow. First, a packet is classified based on the usual 5-tuple flow ID (*i.e.*, <srcIP, dstIP, srcPort, dstPort, proto>). Then, a path-specific classification is conducted. Figure 2 illustrates an overall procedure of updating TIB.

When a packet arrives at a server, we first retrieve its metadata (flow ID, path information (*i.e.*, link IDs) and bytes). Because the path information is irrelevant to the upper layer protocols, we strip it off from the packet header in Open vSwitch (OVS) before it is delivered to the upper stack via the regular route. Next, using the flow ID and link IDs together as a key, we create or update a per-path flow record in trajectory memory. Note that link IDs do not represent a complete end-to-end path yet. Each record contains flow ID, link IDs, packet and byte counts and flow duration. That is, one per-path flow record corresponds to statistics on packets of the same flow that traversed the same path. Thus, at a given point in time, more than one per-path flow record can be associated with a flow. Similar to NetFlow, if FIN or RST packet is seen or a per-path flow record is not updated for a certain time period (*e.g.*, 5 seconds), the flow record is evicted from the trajectory memory and forwarded to the trajectory construction sub-module.

The sub-module then constructs an end-to-end path with link IDs in a per-path flow record. It first looks up the trajectory cache with srcIP and link IDs. If there is a cache hit, it immediately converts the link IDs into a path. If not, the module maps link IDs to a series of switches by referring to a physical topology, and builds an end-to-end path. It then updates the trajectory cache with (srcIP, link IDs, path). In this process, a "static" physical network topology graph suffices, and there is no need for dynamically updating it unless the topology changes physically. Finally, the module writes a record (<flow ID, path, stime, etime, #bytes, #pkts>) to TIB.

We add to OVS about 150 lines of C code to support the trajectory extraction and store function, and run the modified OVS on DPDK [4] for high-speed packet processing (e.g., 10 Gbps). The module is implemented with roughly 600 lines of C++ code. We build TIB using MongoDB [7].

Query processing. PathDump maintains TIB in a distributed fashion (across all servers in the datacenter). The controller sends server agents a query, composed of PathDump APIs (§2.1), which in turn processes the TIB data and returns results to the controller. The querying mechanism is composed of about 640 lines of Python code.


Operator
Server
Alarm (..., POOR_PERF, ...)
On-demand debugging Applications
Controller
Event-driven debugging Applications
execute
Server
Figure 3: Workflow of PathDump.

Depending on debugging applications, the controller needs to consult more than one TIB. For instance, to check path conformance of a packet or flow, accessing only one TIB is sufficient. On the other hand, some debugging queries (*e.g.*, load imbalance diagnosis; see §4.2) need path information from all distributed TIBs.

To handle these different needs properly, we implement two types of query mechanisms: (i) direct query and (ii) multi-level query. The former is a query that is directly sent to one specific TIB by the controller. Inspired by Dremel [27] and iMR [25], we design a multi-level query mechanism whereby the controller creates a multi-level aggregation tree and distributes it alongside a query. When a server receives query and tree, it performs two tasks: (i) query execution on local TIB and (ii) redistribution of both query and tree. The query results are aggregated from the bottom of the tree. However, the current implementation is not fully optimized yet; and improving its efficacy is left as part of our future work.

In general, multi-level data aggregation mechanisms including ours can be ineffective in improving response times when the data size is not large and there is no much data reduction during aggregation along the tree. In §5, we present the tradeoff through two multi-level queries—flow size distribution and top-*k* flows.

Finally, when a query is executed, the latest TIB records relevant to the query may reside in the trajectory memory, yet to be exported to the TIB. We handle this by creating an IPC channel and allowing the server agent to look up the trajectory memory. Not all debugging applications require to access the trajectory memory. Instead, the alerts raised by `Alarm()` trigger the access to the memory for debugging at even finer-grained time scales.

Active monitoring module. Timely triggering of a debugging process requires fast detection of symptoms on network problems. Servers are a right vantage point to instantly sense the symptoms like TCP timeouts, high retransmission rates, large RTT and low throughput.

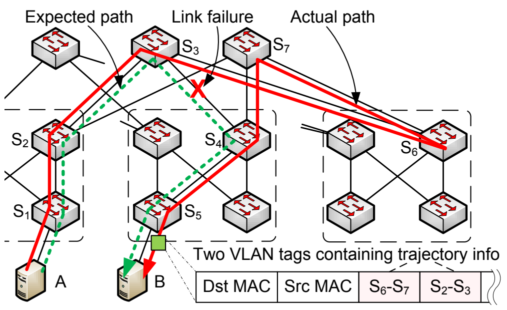


Figure 4: An example of path conformance check. The dotted green line is an expected path and the red line is an actual path that packet traverses.

We thus implement a monitoring module at server that checks TCP connection performance, and promptly raises alerts to the controller in the advent of abnormal TCP behavior. Specifically, by using `tcpretrans` script in perf-tools[5], the module checks the packet retransmission of individual flows at regular intervals (configured by installing a query). If packet retransmissions are observed more than a configured frequency, an alert is raised to the controller, which can subsequently take actions in response. Thus, this active TCP performance monitoring allows fast troubleshooting. We exploit the alert functionality to expedite debugging tasks such as silent packet drop localization (§4.3), blackhole diagnosis (§4.4) and TCP performance anomaly diagnosis (§4.6).

In addition, network behavior desired by operators can be expressed as network invariants (*e.g.*, maximum path length), which can be installed on end-hosts using `install()`. This module uses `Alarm()` to inform any invariant's violation as depicted in §2.3.

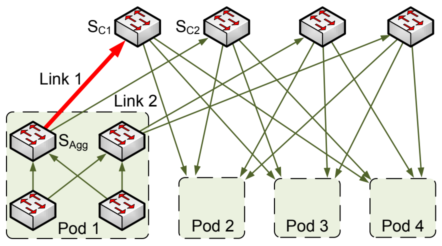
3.3 PathDump controller

PathDump controller plays two roles: installing flow rules on switches and executing debugging applications.

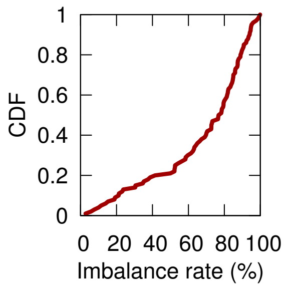
It installs flow rules in switches that append link IDs in the packet header (using `push_vlan` output action) in order to enable packet trajectory tracing. This is one-time task when the controller is initialized, and the rules are not modified once they are installed. We use switches that support a pipeline of flow tables and that are therefore compatible with OpenFlow specification v1.3.0.

Debugging applications can be executed under two contexts as depicted in Figure 3: (i) event-driven, and (ii) on-demand. It is event-driven when the controller receives alerts from the active monitoring module at end-hosts. The other, obvious way is that the operator executes debugging applications on demand. Queries and results are exchanged via direct query or multi-level query. The controller consists of about 650 lines of Python code.

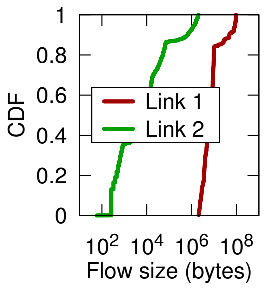
[5]https://github.com/brendangregg/perf-tools



(a) S_{Agg} poorly load-balances traffic



(b) Load imbalance rate



(c) Flow size distribution

Figure 5: Load imbalance diagnosis. (a) illustrates a load imbalance case. (b) shows, as reference, the load imbalance rate between links 1 and 2. (c) shows the flow size distribution built by querying all TIBs.

4 Applications

PathDump can support various debugging applications for datacenter network problems including both persistent and transient ones (see Table 2 in the appendix for a comprehensive list of debugging applications). In this section, we highlight a subset of those applications.

4.1 Path conformance check

A path conformance test is to check whether an actual path taken by a packet conforms to operator policy. To demonstrate that, we create an experimental case shown in Figure 4. In the figure, the intended path of a packet is a 4-hop shortest path from server *A* to *B*. However, a link failure between switches S_3 and S_4 makes S_3 forward the packet to S_6 (we implement a simple failover mechanism in switches with a few flow rules). As a result, the packet ends up traversing a 6-hop path. The PathDump agent in *B* is configured with a predicate, as a query (as depicted in §2.3), that a 6-hop or longer path is a violation of the path conformance policy. The agent detects such packets in real time and alerts the controller to the violation along with the flow key and trajectory.

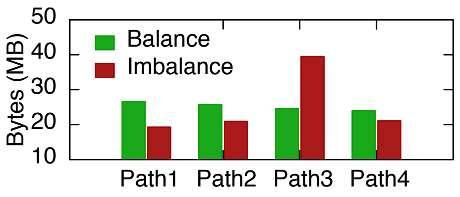


Figure 6: Traffic distribution of a flow along four different paths under balanced and imbalanced cases.

4.2 Load imbalance diagnosis

Datacenter networks employ load-balancing mechanisms such as ECMP and packet spraying [15] to exploit numerous equal-cost paths. However, when these mechanisms work poorly, uneven load splits can hurt throughput and flow completion time. PathDump can help narrow down the root causes of load imbalance problems, which we demonstrate using two load-balancing mechanisms: (i) ECMP and (ii) packet spraying.

ECMP load-balancing. This scenario (Figure 5(a)) assumes that a poor hash function always creates collisions among large flows. For the scenario, we configure switch S_{Agg} in pod 1 such that it splits traffic based on flow size. Specifically, if a flow is larger than 1 MB in size, it is pushed onto link 1. If not, it is pushed onto link 2. Based on the web traffic model in [10], we generate flows from servers in pod 1 to servers in the remaining pods. As a metric, we use imbalance rate, $\lambda = (L_{max}/\overline{L} - 1) \times 100$ (%) where L_{max} is the maximum load on any link and $\overline{L}$ is the mean load over all links [31].

Figure 5(b) shows the load imbalance rate between the two links measured every 5 seconds for 10 minutes. During about 80% of the time, the imbalance rate is 40% or higher. With the load imbalance diagnosis application in §2.3, PathDump issues a multi-level query to all servers and collects byte counts of flows that visited those two links. As shown in Figure 5(c), flow size distributions on the two links are sharply divided around 1 MB. With flow IDs and their sizes in the TIBs, operators can reproduce this load imbalance scenario for further investigation.

This scenario illustrates how PathDump handles a persistent problem. The application can be easily extended for tackling transient ECMP hash collisions among long flows by exploiting the TCP performance alert function.

Packet spraying. In this scenario, packets of a flow are split among four possible equal-cost paths between a source and destination. For demonstration, we create two cases: (i) a balanced case and (ii) an imbalanced case. In a balanced case, the split process is entirely random, thereby ensuring fair load-balance, whereas in an imbalanced case, we configure switches so that more packets are deliberately forwarded to one of the paths (*i.e.*, Path 3 in Figure 6). The flow size is set to 100 MB. Figure 6 is drawn using per-path statistics of the flow obtained from the destination TIB. As shown in the figure, operators can check whether packet spraying works well or not. In case of poor load-balancing, they can tell which path (more precisely, which link) is under- or over-utilized. The per-packet path tracing ability of PathDump allows this level of detailed analysis. For real-time monitoring, it is sufficient to install a query (using `install()`) that monitors the traffic amount difference among subflows.

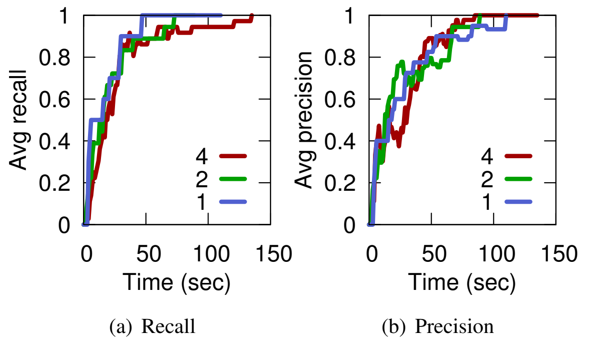


Figure 7: Performance of the silent random packet drop debugging algorithm. Average recall and precision are presented over 10 runs. The network load is set to 70% and each faulty interface drops packets at 1% rate. The numbers (i.e., 1, 2 and 4) in legend denote the number of faulty interfaces.

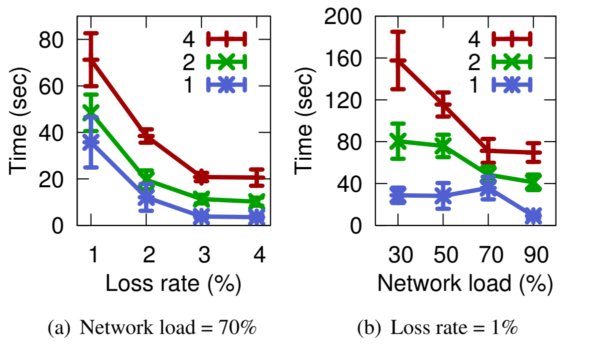
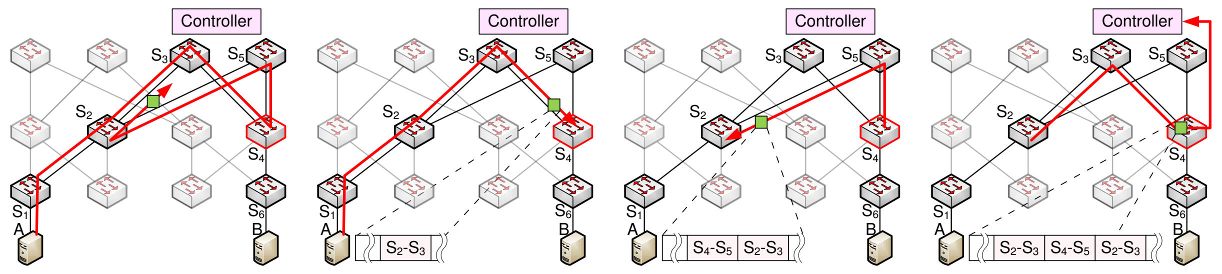


Figure 8: Time taken to reach 100% recall and precision. The numbers (i.e., 1, 2 and 4) in legend denote the number of faulty interfaces. The error bar is standard error, i.e., $\sigma/\sqrt{n}$ where σ is standard deviation and n is the number of runs (= 10).

4.3 Silent random packet drops

We implement the silent packet drop debugging application as described in §2.3 and conduct experiments in a 4-ary fat-tree topology, where each end-host generates traf-



(a) A routing loop case (b) Step 1 (c) Step 2 (d) Step 3

Figure 9: Debugging a routing loop. (a) A routing loop is illustrated. (b) A packet carries a VLAN tag whose value is an ID for link $S_2 - S_3$ appended by S_3. (c) S_4 bounces the packet to S_5; S_5 forwards the packet to one remaining egress port (to S_2) while appending an ID for link $S_4 - S_5$ to the packet header. (d) S_3 appends a third tag of which the value is a ID for link $S_2 - S_3$; at S_4, the packet is *automatically* forwarded to the controller since ASIC in switches only recognizes two VLAN tags whilst the packet carries three; at this stage, the controller immediately detects the loop by finding the repeated link $S_2 - S_3$ from the packet header.

fic based on the same web traffic model. We configure 1-4 randomly selected interfaces such that they drop packets at random. We run the MAX-COVERAGE algorithm and evaluate its performance based on two metrics: recall and precision. Recall is $\frac{\#TPs}{\#TPs+\#FNs}$ while precision is $\frac{\#TPs}{\#TPs+\#FPs}$ where true positive is denoted as TP, false negative as FN, and false positive as FP.

In our experiment, as time progresses, the number of alerts received by the controller increases; so does the number of failure signatures. Hence, from Figure 7, we observe the accuracy (both recall and precision) also increases accordingly; the recall increases faster than the precision. It is clear from Figure 8, as loss rate or network load increase, the controller receives alerts from end-hosts at higher rate, and thus the algorithm takes less time to obtain 100% recall and precision, making it possible to debug the silent random packet drops fast and accurately.

4.4 Blackhole diagnosis

We demonstrate how PathDump reduces a debugging search space with a blackhole scenario in the network with a 4-ary fat-tree topology where packet spraying is deployed. Again, we generate the same background traffic used in §4.3 to create noises in the debugging process. We create a 100 KB TCP flow and its packets are randomly routed through four possible paths and test two cases.

Blackhole at an aggregate-core link. Obviously, the subflow traffic passing the blackhole link is all dropped. The controller receives an alarm from PathDump agent at sender in 1 sec, immediately retrieves all TIB records for the flow and finds one record for the dropped subflow missing. While examining the paths found in TIB records, it finds that one path did not appear in the TIB. Since only one path (hence, one subflow) was impacted, it produces three switches as a potential culprit: core switch, source and destination aggregate switches (thus avoiding the search of all 10 switches in the four paths).

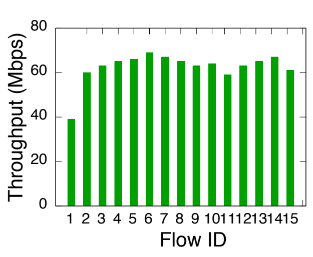
Blackhole at a ToR-aggregate link in the source pod. This blackhole impacts two subflows. The controller identifies two paths that impacted the two subflows using the same way as before. By joining the two paths, the controller can pick four common switches, which should be examined with higher priority.

Note that if more number of flows (and their subflows) are impacted by the blackhole, PathDump can localize the exact source of the blackhole.

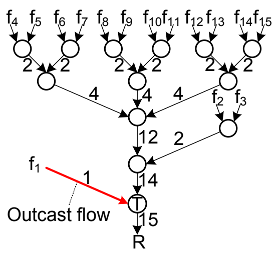
4.5 Routing loop debugging

PathDump debugs routing loop in *real-time* by trapping a suspiciously long path in the network. As discussed in §3.1, a packet carrying more than two tags is automatically directed to the controller. This feature is a foundation of *making routing loops naturally manifest themselves* at the controller. More importantly, the fact that the controller has a direct control over suspicious packets makes it possible to detect routing loops *of any size*.

Real timeliness. We create a 4-hop routing loop as shown in Figure 9(a). Specifically, switch S_4 is misconfigured and all core switches are configured to choose an alternative egress port except the ingress port of a packet. In the figure, switches from S_2 to S_5 constitute the loop. Under this setup, it takes about **47 ms** on average until the controller detects the loop. When the packet trapped in this loop ends up carrying three tags (see Figures 9(b)–9(d)) and appears at the controller, two of the tags have the same link ID ($S_2 - S_3$ in Figure 9(d)). Hence, the loop is detected immediately at this stage.



(a)



(b)

Figure 10: Diagnosis of TCP outcast. Unfairness of throughput is shown in (a). In (b), the communication graph is mapped onto a physical topology, and edge weight is the number of flows arriving at an input port. Both data sets are made available from TIB.

Detecting loops of any size. In this scenario, we create a 6-hop routing loop (not shown for brevity). The controller finds no repeated link IDs from three tags when it sees the packet for the first time. The controller locally stores the three tags, strips them off from the packet header, and sends the packet back to the switch. Since the packet is trapped in the 6-hop loop, it will have another set of three tags and be forwarded to the controller. This time, comparing link IDs in previous and current tags, the controller observes that there is at least one repeated link ID and detects the loop. The whole process took ∼**115 ms**. Detecting even larger loops involves exactly the same procedure.

4.6 TCP performance anomaly diagnosis

PathDump can diagnose incast [12] and outcast [32] problems in a fine-grained manner although they are transient. In particular, we test a TCP outcast scenario. For a realistic setup, we generate the same type of TCP background traffic used in §4.4. In addition to that, 15 TCP senders send data to a single receiver for 10 seconds. Thus, as shown in Figure 10(b), a flow from f_1 and 14 flows from $f_2 - f_{15}$ arrive on two different input ports at switch T. They compete for the same output port at the switch toward receiver R. As a result, these flows experience the port blackout phenomenon, and the flow from f_1 sees the most throughput loss (see [32] for more details).

Every 200 ms (default TCP timeout value) the server agents run a query that generates alerts when their TCP flows repeatedly retransmit packets. The diagnosis application at the controller starts to work when it sees a minimum of 10 alerts from different sources to a particular destination. Since all alerts specify R as receiver, the application requests flow statistics (*i.e.*, bytes, path) from R and diagnoses the root cause for high alerts. It first analyzes the throughput for each sender (Figure 10(a)) and constructs a path tree for all 15 flows (Figure 10(b)). It then identifies that the flow from f_1 (one closest to the receiver) is most highly penalized. PathDump concludes the TCP unfairness stems from the outcast because these patterns fit the outcast's profile. We observe that the application initiates its diagnosis in 2-3 seconds since the onset of flows and finishes it within next 200 ms.

5 System Evaluation

We first study the performance of direct and multi-level queries in terms of response time and data overheads. We then evaluate CPU and memory overheads at end-host in processing packet stream and in executing queries.

5.1 Experimental setup

We build a real testbed that consists of 28 physical servers; each server is equipped with Xeon 4-core 3.1 GHz CPU and a dual-port 1 GbE card. Using the two interfaces, we separate management channel from data channel. The controller and servers communicate with each other through the management channel to execute queries. Each server runs four docker containers (in total, 112 containers). Each container is assigned one core and runs a PathDump agent to access TIB in it. In this way, we test up to 112 TIBs (*i.e.*, 112 end-hosts). We only refer to container as end-host during the query performance evaluation. Each TIB has 240K flow entries, which roughly corresponds to the number of flows seen at a server for about an hour. We estimate the number based on the observation that average flow inter-arrival time seen at server is roughly 15 ms (∼67 flows/sec) [19].

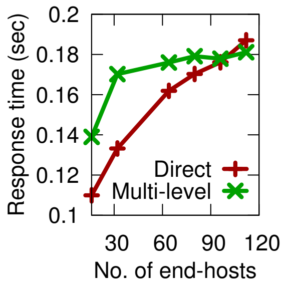
For multi-level query execution, we construct a logical 4-level aggregation tree with 112 end-hosts. Our PathDump controller sits on the top of the tree (level 0). Right beneath the controller are 7 nodes or end-hosts (level 1). Each first-level node has, as its child, four nodes (level 2), each of which has four nodes at the bottom (level 3).

For the packet progressing overhead experiment, we use another server equipped with a 10 GbE card. In this test, we forward packets from all other servers to a virtual port in DPDK vSwitch via the physical 10GbE NIC.

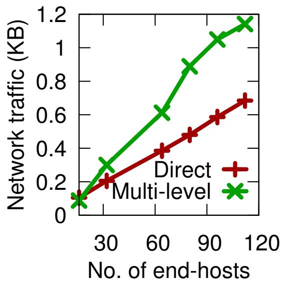
5.2 Query performance

We compare the performance of direct query with that of multi-level query. To understand which type of query suits well to a debugging application, we measure two key metrics: i) end-to-end response time, and ii) total data volume generated. We test two queries—flow size distribution of a link and top-k flows. For the top-k flows query, we set k to 10,000. Results are averaged over 20 runs.

Results. Through these experiments, we make two observations (confirmed via Figures 11 and 12) as follows.

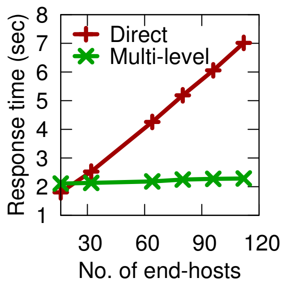


(a) Response time

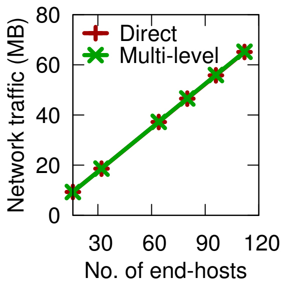


(b) Traffic amount

Figure 11: Average end-to-end response time and traffic amount of a flow size distribution query.



(a) Response time

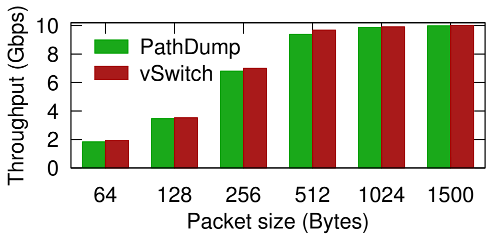


(b) Traffic amount

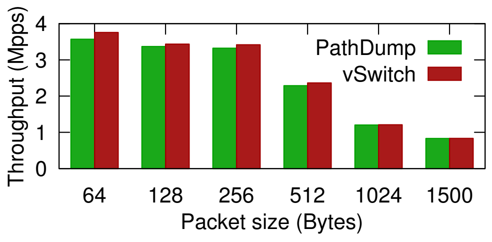
Figure 12: Average end-to-end response time and traffic amount of a top-$10,000$ flows query.

1) When more servers are involved in a query, multi-level query is in general better than direct query. Figure 11(a) shows that multi-level query initially takes longer than direct query. However, the response time gap between the two gets smaller as the number of servers increases. This is due to three reasons. First, the aggregation time (the time to aggregate responses at the controller) of direct query is always larger than that of multi-level query. Second, the aggregation time of direct query linearly grows in proportion to the number of end-hosts whereas that of multi-level query gradually grows. Lastly, network delays of both queries change little regardless of the number of servers.

2) If aggregation reduces response data amount substantially, multi-level query is more efficient than direct query. When multi-level query is employed for computing the top-k flows, $(n_i - 1) \cdot k$ number of key-value pairs are discarded at level $i-1$ during aggregation where n_i is the number of nodes at level i ($i < 3$). A massive data reduction occurs through the aggregation tree. Hence, the data amount exchanged in multi-level query is similar to that in direct query (Figure 12(b)). Moreover, the computation overhead for aggregation is distributed across multiple intermediate servers. On the contrary, in direct query, the controller alone has to process a large number of key-value pairs (*i.e.*, $k \cdot n_3$ where n_3 is the total number of servers used). Hence, the majority of the response time is attributed to computation at the controller, and the response time grows linearly as the number of servers increases (Figure 12(a)). Due to the horizontal scaling nature of multi-level query, its response times remain steady regardless of the number of servers. In summary, these results suggest that multi-level query can scale well even for a large cluster and direct query is recommended when a small number of servers are queried.



(a) Throughput in Gbits per second



(b) Throughput in million-packets per second

Figure 13: Forwarding throughput of PathDump and vSwitch. Each bar represents an average over 30 runs.

5.3 Overheads

Packet processing. We generate traffic by varying its packet size from 64 to 1500 bytes. Each packet carries 1-2 VLAN tags. While keeping about 4K flow records (roughly equivalent to 100K flows/sec at a rack switch connected to 24 hosts) in the trajectory memory, PathDump does about 0.8–3.6M lookups/updates per second (0.8M for 1500B packets and 3.6M for 64B). Under these conditions, we measure average throughput in terms of bits and packets per second over 30 runs.

From Figure 13, we observe that PathDump introduces a maximum of 4% throughput loss compared to the performance of the vanilla DPDK vSwitch. The figure omits confidence intervals as they are small. In all cases, the throughput difference is marginal. Note that due to the limited CPU and memory resources allocated, DPDK vSwitch itself suffers throughput degradation as packet size decreases. Nevertheless, it is clear that PathDump introduces minimal packet processing overheads atop DPDK vSwitch.

Query processing. We measure CPU resource demand for continuous query processing at end-host. The controller generates a mix of direct and multi-level queries continuously in a serialized fashion (i.e., a new query after receiving response for previous one). We observe that less than 25% of one core cycles is consumed at end-host. As datacenter servers are equipped with multi-core CPUs (e.g., 18-core Xeon E5-2699 v3 processor), the query processing introduces relatively less overheads.

Storage. PathDump only needs about 10 MB of RAM at a server for packet trajectory decoding, trajectory memory and trajectory cache. It also needs about 110 MB of disk space to store 240K flow entries (roughly equivalent to an hour's worth of flows observed at a server).

6 Related Work

There has been a tremendous recent effort in building tools for efficient management and debugging of tasks. Each tool works at a unique operating point between supported classes of network debugging problems, accuracy, network bandwidth overheads, and desired functionality from network elements. We summarize the most related of these tools below.

Generality. Several tools support a fairly general class of network debugging problems with high accuracy — PathQuery [30], NetSight [17], NetPlumber [20], VeriFlow [22] and several other systems [21, 26]. However, these systems make arguably strong tradeoffs to achieve generality with accuracy. In particular, for many network debugging problems, these systems [20, 21, 22, 26] require a snapshot of the entire data-plane state and may only be able to capture events at coarse-grained time-scales. Netsight [17] captures per-packet per-switch log for out-of-band analysis; capturing per-packet per-switch logs leads to very high bandwidth requirements and out-of-band analysis typically leads to high latency between the time of occurrence of an event and when the event is diagnosed. Finally, PathQuery [30] supports network debugging by dynamically installing switch rules and using SQL-like queries on these switches; this not only requires dynamic installation of switch rules and large amount of data plane resources to achieve generality but also debugging at coarse-grained time-scales. PathDump, by pushing much of the debugging functionality to the end-hosts, makes a different tradeoff — it gives up on a small class of network debugging problems, but alleviates the overheads of dynamic switch rule installation, per-packet per-switch log generation and periodic data plane snapshots.

Accuracy. Several recent proposals alleviate the overheads of aforementioned systems using sampling [8, 16, 24, 33, 35, 39, 41], mirroring of sampled packets [33, 41], active packet probes [9, 40, 41], and a potpourri of these techniques [41]. These tools have two main limitations: (i) they make the functionality implemented at the network elements (precisely the elements that these tools are trying to debug) even more complex; and (ii) sampling and/or active probing, by definition, leads to missed network events (low accuracy). In contrast, PathDump avoids complex operations like packet sampling, packet mirroring, and/or active probing, by pushing much of the network debugging functionality to the end-hosts. PathDump, thus, performs debugging with high accuracy at finer-grained time-scales without incurring overheads.

End-host based tools. Several recent proposals have advocated to move the functionality to the edge devices [11, 29, 38]. SNAP [38] logs events (e.g., TCP statistics and socket-calls) at end-hosts to infer network problems. Felix [11] proposed a declarative query language for end-host based network measurement. Finally, independent to our work, Trumpet [29] proposes to push the debugging functionality to the end-hosts. PathDump differs from and complements these systems along several dimensions. First, the core idea of PathDump is to exploit the packet trajectories to debug a large class of network problems; capturing and utilizing packet trajectories for debugging purposes complements the techniques used in above tools. Second, in addition to the monitoring functionality of Trumpet [29], PathDump also allows the network operators to slice-and-dice the captured logs to debug a network problem.

7 Conclusion

This paper presents PathDump, a network debugger that partitions the debugging functionality between the edge devices and the network switches (in contrast to an entirely in-network implementation used in existing tools). PathDump does not require network switches to perform complex operations like dynamic switch rule updates, per-packet per-switch log generation, packet sampling, packet mirroring, etc., and yet helps debug a large class of network problems over fine-grained time-scales. Evaluation of PathDump over operational network testbeds comprising of commodity network switches and end-hosts show that PathDump requires minimal data plane resources (*e.g.*, switch rules and packet header space) and end-host resources (*e.g.*, CPU and memory).

Application	**Description**	**PathDump**	**PathQuery**[30]	**Everflow**[41]	**NetSight**[17]	**TPP**[18]
Loop freedom [17]	Detect forwarding loops	✓	✓	✓	✓	?
Load imbalance diagnosis [41]	Get fine-grained statistics of all flows on set of links	✓	✓	✓	✓	✓
Congested link diagnosis [30]	Find flows using a congested link, to help rerouting	✓	✓	✓	✓	✓
Silent blackhole detection [41, 30]	Find switch that drops all packets silently	✓	✓	✓	✓	✗
Silent packet drop detection [41]	Find switch that drops packets silently and randomly	✓	✓	✓	✓	✗
Packet drops on servers [41]	Localize packet drop sources (network vs. server)	✓	✓	✓	✓	✓
Overlay loop detection [41]	Loop between SLB and physical IP	✗	✓	✓	✓	?
Protocol bugs [41]	Bugs in the implementation of network protocols	✓	✓	✓	✓	?
Isolation [17]	Check if hosts are allowed to talk	✓	✓	✓	✓	✓
Incorrect packet modification [17]	Localize switch that modifies packet incorrectly	✗	✓	?	✓	✗
Waypoint routing [17, 30]	Identify packets not passing through a waypoint	✓	✓	✓	✓	✓
DDoS diagnosis [30]	Get statistics of DDoS attack sources	✓	✓	✓	✓	✓
Traffic matrix [30]	Get traffic volume between all switch pairs in a switch	✓	✓	✓	✓	✓
Netshark [17]	Nework-wide path-aware packet logger	✓	✓	✓	✓	✓
Max path length [17]	No packet should exceed path length of size n	✓	✓	✓	✓	✓

Table 2: Debugging applications supported by existing tools and PathDump. The table assumes that Everflow performs per-switch per-packet mirroring. Of course, this will have much higher bandwidth requirements than the network traffic itself. If Everflow uses the proposed sampling to minimize bandwidth overheads, many of the above applications will not be supported by Everflow.

Acknowledgments

We would like to thank anonymous reviewers and our shepherd George Porter for their feedback and suggestions. We would also like to thank Ratul Mahajan for many interesting discussions. This work was in part supported by EPSRC grant EP/L02277X/1.

Appendix

Table 2 summarizes the set of applications discussed in several recent papers, and outlines whether a tool supports an application or not (the table intentionally ignores the resource requirements and/or complexity of supporting each individual application for the respective tools).

References

[1] Amazon EBS failure brings down Reddit, Imgur, others. http://tinyurl.com/oxmugps.

[2] Amazon.com suffers outage: Nearly $5m down the drain? http://tinyurl.com/od7vhm8.

[3] Azure outage raises questions about public cloud for mission-critical apps. http://tinyurl.com/no92ojy.

[4] DPDK: Data Plane Development Kit. http://dpdk.org/.

[5] Flask. http://flask.pocoo.org/.

[6] Google outage: Internet traffic plunges 40%. http://tinyurl.com/l7hegn6.

[7] MongoDB. https://www.mongodb.org/.

[8] Sampled NetFlow. http://www.cisco.com/c/en/us/td/docs/ios/12_0s/feature/guide/12s_sanf.html, 2003.

[9] K. Agarwal, E. Rozner, C. Dixon, and J. Carter. SDN Traceroute: Tracing SDN Forwarding Without Changing Network Behavior. In *ACM HotSDN*, 2014.

[10] M. Alizadeh, S. Yang, M. Sharif, S. Katti, N. McKeown, B. Prabhakar, and S. Shenker. pFabric: minimal near-optimal datacenter transport. In *ACM SIGCOMM*, 2013.

[11] H. Chen, N. Foster, J. Silverman, M. Whittaker, B. Zhang, and R. Zhang. Felix: Implementing traffic measurement on end hosts using program analysis. In *ACM SIGCOMM SOSR*, 2016.

[12] Y. Chen, R. Griffith, J. Liu, R. H. Katz, and A. D. Joseph. Understanding TCP Incast Throughput Collapse in Datacenter Networks. In *ACM Workshop on Research on Enterprise Networking*, 2009.

[13] R. Cole, K. Ost, and S. Schirra. Edge-Coloring Bipartite Multigraphs in O(E log D) Time. *Combinatorica*, 21(1), 2001.

[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*, 2011.

[15] A. Dixit, P. Prakash, Y. C. Hu, and R. R. Kompella. On the Impact of Packet Spraying in Data Center Networks. In *IEEE INFOCOM*, 2013.

[16] N. G. Duffield and M. Grossglauser. Trajectory Sampling for Direct Traffic Observation. *IEEE/ACM ToN*, 9(3), 2001.

[17] N. Handigol, B. Heller, V. Jeyakumar, D. Mazières, and N. McKeown. I Know What Your Packet Did Last Hop: Using Packet Histories to Troubleshoot Networks. In *USENIX NSDI*, 2014.

[18] V. Jeyakumar, M. Alizadeh, Y. Geng, C. Kim, and D. Mazières. Millions of Little Minions: Using Packets for Low Latency Network Programming and Visibility. In *ACM SIGCOMM*, 2014.

[19] S. Kandula, S. Sengupta, A. Greenberg, P. Patel, and R. Chaiken. The Nature of Data Center Traffic: Measurements & Analysis. In *ACM IMC*, 2009.

[20] 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*, 2013.

[21] P. Kazemian, G. Varghese, and N. McKeown. Header Space Analysis: Static Checking for Networks. In *USENIX NSDI*, 2012.

[22] A. Khurshid, X. Zou, W. Zhou, M. Caesar, and P. B. Godfrey. VeriFlow: Verifying Network-Wide Invariants in Real Time. In *USENIX NSDI*, 2013.

[23] R. R. Kompella, J. Yates, A. Greenberg, and A. C. Snoeren. Detection and localization of network black holes. In *IEEE INFOCOM*, 2007.

[24] Y. Li, R. Miao, C. Kim, and M. Yu. FlowRadar: A Better NetFlow for Data Centers. In *USENIX NSDI*, 2016.

[25] D. Logothetis, C. Trezzo, K. C. Webb, and K. Yocum. In-situ MapReduce for Log Processing. In *USENIX ATC*, 2011.

[26] H. Mai, A. Khurshid, R. Agarwal, M. Caesar, P. B. Godfrey, and S. T. King. Debugging the Data Plane with Anteater. In *ACM SIGCOMM*, 2011.

[27] S. Melnik, A. Gubarev, J. J. Long, G. Romer, S. Shivakumar, M. Tolton, and T. Vassilakis. Dremel: Interactive analysis of web-scale datasets. In *VLDB*, 2010.

[28] M. Moshref, M. Yu, R. Govindan, and A. Vahdat. Dream: dynamic resource allocation for software-defined measurement. In *ACM SIGCOMM*, 2014.

[29] M. Moshref, M. Yu, R. Govindan, and A. Vahdat. Trumpet: Timely and Precise Triggers in Data Centers. In *ACM SIGCOMM*, 2016.

[30] S. Narayana, M. Tahmasbi, J. Rexford, and D. Walker. Compiling Path Queries. In *USENIX NSDI*, 2016.

[31] O. Pearce, T. Gamblin, B. R. de Supinski, M. Schulz, and N. M. Amato. Quantifying the Effectiveness of Load Balance Algorithms. In *ACM ICS*, 2012.

[32] P. Prakash, A. Dixit, Y. C. Hu, and R. Kompella. The TCP Outcast Problem: Exposing Unfairness in Data Center Networks. In *USENIX NSDI*, 2012.

[33] J. Rasley, B. Stephens, C. Dixon, E. Rozner, W. Felter, K. Agarwal, J. Carter, and R. Fonseca. Planck: Millisecond-scale monitoring and control for commodity networks. In *ACM SIGCOMM*, 2014.

[34] A. Singh et al. Jupiter Rising: A Decade of Clos Topologies and Centralized Control in Google's Datacenter Network. In *ACM SIGCOMM*, 2015.

[35] J. Suh, T. T. Kwon, C. Dixon, W. Felter, and J. B. Carter. OpenSample: A Low-Latency, Sampling-Based Measurement Platform for Commodity SDN. In *IEEE ICDCS*, 2014.

[36] P. Tammana, R. Agarwal, and M. Lee. CherryPick: Tracing Packet Trajectory in Software-defined Datacenter Networks. In *ACM SIGCOMM SOSR*, 2015.

[37] A. Wundsam, D. Levin, S. Seetharaman, and A. Feldmann. OFRewind: Enabling Record and Replay Troubleshooting for Networks. In *USENIX ATC*, 2011.

[38] 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 *USENIX NSDI*, 2011.

[39] M. Yu, L. Jose, and R. Miao. Software defined traffic measurement with OpenSketch. In *USENIX NSDI*, 2013.

[40] H. Zeng, P. Kazemian, G. Varghese, and N. McKeown. Automatic Test Packet Generation. *IEEE/ACM ToN*, 22(2):554–566, 2014.

[41] Y. Zhu, N. Kang, J. Cao, A. Greenberg, G. Lu, R. Mahajan, D. Maltz, L. Yuan, M. Zhang, B. Y. Zhao, and H. Zheng. Packet-Level Telemetry in Large Datacenter Networks. In *ACM SIGCOMM*, 2015.

Network Requirements for Resource Disaggregation

Peter X. Gao *UC Berkeley* · *Akshay Narayan* *UC Berkeley* · *Sagar Karandikar* *UC Berkeley* · *Joao Carreira* *UC Berkeley* · *Sangjin Han* *UC Berkeley*

Rachit Agarwal *Cornell University* · *Sylvia Ratnasamy* *UC Berkeley* · *Scott Shenker* *UC Berkeley/ICSI*

Abstract

Traditional datacenters are designed as a collection of servers, each of which tightly couples the resources required for computing tasks. Recent industry trends suggest a paradigm shift to a disaggregated datacenter (DDC) architecture containing a pool of resources, each built as a standalone resource blade and interconnected using a network fabric.

A key enabling (or blocking) factor for disaggregation will be the network – to support good application-level performance it becomes critical that the network fabric provide low latency communication even under the increased traffic load that disaggregation introduces. In this paper, we use a workload-driven approach to derive the minimum latency and bandwidth requirements that the network in disaggregated datacenters must provide to avoid degrading application-level performance and explore the feasibility of meeting these requirements with existing system designs and commodity networking technology.

1 Introduction

Existing datacenters are built using servers, each of which tightly integrates a small amount of the various resources needed for a computing task (CPU, memory, storage). While such server-centric architectures have been the mainstay for decades, recent efforts suggest a forthcoming paradigm shift towards a *disaggregated* datacenter (DDC), where each resource type is built as a standalone resource "blade" and a network fabric interconnects these resource blades. Examples of this include Facebook Disaggregated Rack [8], HP "The Machine" [13], Intel Rack Scale Architecture [19], SeaMicro [24] as well as prototypes from the computer architecture community [31,46,51].

These industrial and academic efforts have been driven largely by hardware architects because CPU, memory and storage technologies exhibit significantly different trends in terms of cost, performance and power scaling [10,21,23,60]. This, in turn, makes it increasingly hard to adopt evolving resource technologies within a server-centric architecture (*e.g.*, the memory-capacity wall making CPU-memory co-location unsustainable [62]). By decoupling these resources, DDC makes it easier for each resource technology to evolve independently and reduces the time-to-adoption by avoiding the burdensome process of redoing integration and motherboard design.[1] In addition, disaggregation also enables fine-grained and efficient provisioning and scheduling of individual resources across jobs [40].

A key enabling (or blocking) factor for disaggregation will be the network, since disaggregating CPU from memory and disk requires that the inter-resource communication that used to be contained *within* a server must now traverse the network fabric. Thus, to support good application-level performance it becomes critical that the network fabric provide low latency communication for this increased load. It is perhaps not surprising then that prototypes from the hardware community [8,13,19,24,31,46,51] all rely on new high-speed network components – e.g., silicon photonic switches and links, PCIe switches and links, new interconnect fabrics, etc. The problem, however, is that these new technologies are still a long way from matching existing commodity solutions with respect to cost efficiency, manufacturing pipelines, support tools, and so forth. Hence, at first glance, disaggregation would appear to be gated on the widespread availability of new networking technologies.

But are these new technologies strictly *necessary* for disaggregation? Somewhat surprisingly, despite the many efforts towards and benefits of resource disaggregation, there has been little systematic evaluation of the network requirements for disaggregation. In this paper, we take a first stab at eval-

[1]We assume partial CPU-memory disaggregation, where each CPU has some local memory. We believe this is a reasonable intermediate step toward full CPU-memory disaggregation.

uating the *minimum* (bandwidth and latency) requirements that the network in disaggregated datacenters must provide. We define the minimum requirement for the network as that which allows us to maintain application-level performance close to server-centric architectures; i.e., at minimum, we aim for a network that keeps performance degradation small for current applications while still enabling the aforementioned qualitative benefits of resource disaggregation.

Using a combination of emulation, simulation, and implementation, we evaluate these minimum network requirements in the context of ten workloads spanning seven popular open-source systems — Hadoop, Spark, GraphLab, Timely dataflow [26, 49], Spark Streaming, memcached [20], HERD [42], and SparkSQL. We focus on current applications such as the above because, as we elaborate in §3, they represent the worst case in terms of the application *degradation* that may result from disaggregation. Our key findings are:

- Network bandwidth in the range of 40 – 100Gbps is sufficient to maintain application-level performance within 5% of that in existing datacenters; this is easily in reach of existing switch and NIC hardware.
- Network latency in the range of 3 – 5μs is needed to maintain application-level performance. This is a challenging task. Our analysis suggests that the primary latency bottleneck stems from network software rather than hardware: we find the latency introduced by the endpoint is roughly 66% of the inter-rack latency and roughly 81% of the intra-rack latency. Thus many of the switch hardware optimizations (such as terabit links) pursued today can optimize only a small fraction of the overall latency budget. Instead, work on bypassing the kernel for packet processing and NIC integration [33] could significantly impact the feasibility of resource disaggregation.
- We show that the root cause of the above bandwidth and latency requirements is the application's memory bandwidth demand.
- While most efforts focus on disaggregating at the rack scale, our results show that for some applications, disaggregation at the datacenter scale is feasible.
- Finally, our study shows that transport protocols frequently deployed in today's datacenters (TCP or DCTCP) fail to meet our target requirements for low latency communication with the DDC workloads. However, some recent research proposals [30, 36] do provide the necessary end-to-end latencies.

Taken together, our study suggests that resource disaggregation need not be gated on the availability of new networking hardware: instead, minimal performance degradation can be achieved with existing network hardware (either commodity, or available shortly).

Communication	**Latency (ns)**	**Bandwidth (Gbps)**
CPU – CPU	10	500
CPU – Memory	20	500
CPU – Disk (SSD)	10^4	5
CPU – Disk (HDD)	10^6	1

Table 1: Typical latency and peak bandwidth requirements within a traditional server. Numbers vary between hardware.

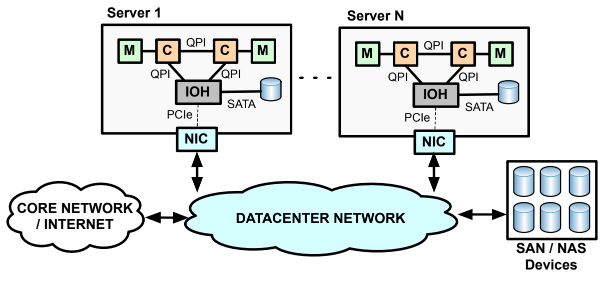
There are two important caveats to this. First, while we may not need network changes, we will need changes in hosts, for which RDMA and NIC integration (for hardware) and pFabric or pHost (for transport protocols) are promising directions. Second, our point is not that new networking technologies are not worth pursuing but that the adoption of disaggregation *need not be coupled* to the deployment of these new technologies. Instead, early efforts at disaggregation can begin with existing network technologies; system builders can incorporate the newer technologies when doing so makes sense from a performance, cost, and power standpoint.

Before continuing, we note three limitations of our work. First, our results are based on ten specific workloads spanning seven open-source systems with varying designs; we leave to future work an evaluation of whether our results generalize to other systems and workloads.[2] Second, we focus primarily on questions of network design for disaggregation, ignoring many other systems questions (*e.g.*, scheduler designs or software stack) modulo discussion on understanding latency bottlenecks. However, if the latter does turn out to be the more critical bottleneck for disaggregation, one might view our study as exploring whether the network can "get out of the way" (as often advocated [37]) even under disaggregation. Finally, our work looks ahead to an overall system that does not yet exist and hence we must make assumptions on certain fronts (e.g., hardware design and organization, data layout, etc.). We make what we believe are sensible choices, state these choices explicitly in §2, and to whatever extent possible, evaluate the sensitivity of these choices on our results. Nonetheless, our results are dependent on these choices, and more experience is needed to confirm their validity.

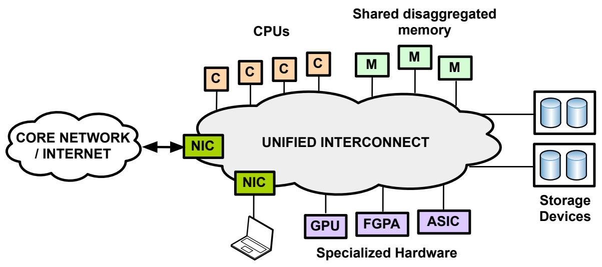
2 Disaggregated Datacenters

Figure 1 illustrates the high-level idea behind a disaggregated datacenter. A DDC comprises standalone hardware "blades"

[2]We encourage other researchers to extend the evaluation with our emulator. https://github.com/NetSys/disaggregation



(a) Current datacenter



(b) Disaggregated datacenter

Figure 1: High-level architectural differences between server-centric and resource-disaggregated datacenters.

for each resource type, interconnected by a network fabric. Multiple prototypes of disaggregated hardware already exist — Intel RSA [19], HP "The Machine" [13], Facebook's Disaggregated Rack [8], Huawei's DC3.0 [12], and SeaMicro [24], as well as research prototypes like FireBox [31], soNUMA [51], and memory blades [46]. Many of these systems are proprietary and/or in the early stages of development; nonetheless, in our study we draw from what information is publicly available to both borrow from and critically explore the design choices made by existing hardware prototypes.

In this section, we present our assumptions regarding the hardware (§2.1) and system (§2.2) architecture in a disaggregated datacenter. We close the section by summarizing the key open design choices that remain after our assumptions (§2.3); we treat these as design "knobs" in our evaluation.

2.1 Assumptions: Hardware Architecture

Partial CPU-memory disaggregation. In general, disaggregation suggests that each blade contains one particular resource with a direct interface to the network fabric (Fig. 1). One exception to this strict decoupling is CPU blades: each CPU blade retains some amount of *local* memory that acts as a cache for remote memory dedicated for cores on that blade[3]. Thus, CPU-memory disaggregation can be viewed as expanding the memory hierarchy to include a remote level, which all CPU blades share.

This architectural choice is reported in prior work [12, 31, 46, 47]. While we assume that partial CPU-memory disaggregation will be the norm, we go a step further and evaluate how the amount of local memory impacts *network* requirements in terms of network bandwidth and latency, and transport-layer flow completion times.

Cache coherence domain is limited to a single compute blade. As articulated by others [12, 13, 31], this has the important implication that CPU-to-CPU cache coherence traffic does not hit the network fabric. While partial CPU-memory disaggregation reduces the traffic hitting the network, cache coherence traffic can not be cached and hence directly impacts the network. This assumption is necessary because an external network fabric is unlikely to support the latency and bandwidth requirements for inter-CPU cache coherence (Table 1).

Resource Virtualization. Each resource blade must support virtualization of its resources; this is necessary for resources to be logically aggregated into higher-level abstractions such as VMs or containers. Virtualization of IO resources is widely available even today: many IO device controllers now support virtualization via PCIe, SR-IOV, or MR-IOV features [41] and the same can be leveraged to virtualize IO resources in DDC. The disaggregated memory blade prototyped by Lim et al. [46] includes a controller ASIC on each blade that implements address translation between a remote CPU's view of its address space and the addressing used internally within the blade. Other research efforts assume similar designs. We note that while the implementation of such blades may require some additional new hardware, it requires no change to existing components such as CPUs, memory modules, or storage devices themselves.

Scope of disaggregation. Existing prototypes limit the scope of disaggregation to a very small number of racks. For example, FireBox [31] envisions a single system as spanning approximately three racks and assumes that the *logical* aggregation and allocation of resources is similarly scoped; i.e., the resources allocated to a higher-level abstraction such as a VM or a container are selected from a single FireBox. Similarly, the scope of disaggregation in Intel's RSA is a single rack [19]. In contrast, in a hypothetical datacenter-scale disaggregated system, resources assigned to (for example) a single VM could be selected from anywhere in the datacenter.

Network designs. Corresponding to their assumed scope

[3]We use "remote memory" to refer to the memory located on a standalone memory blade.

Class	Application Domain	Application	System	Dataset
Class A	Off-disk Batch	WordCount	Hadoop	Wikipedia edit history [27]
	Off-disk Batch	Sort	Hadoop	Sort benchmark generator
	Graph Processing	Collaborative Filtering	GraphLab	Netflix movie rating data [22]
	Point Queries	Key-value store	Memcached	YCSB
	Streaming Queries	Stream WordCount	Spark Streaming	Wikipedia edit history [27]
Class B	In-memory Batch	WordCount	Spark	Wikipedia edit history [27]
	In-memory Batch	Sort	Spark	Sort benchmark generator
	Parallel Dataflow	Pagerank	Timely Dataflow	Friendster Social Network [9]
	In-memory Batch	SQL	Spark SQL	Big Data Benchmark [6]
	Point Queries	Key-value store	HERD	YCSB

Table 2: Applications, workloads, systems and datasets used in our study. We stratify the classes in Section 3.

of disaggregation, existing prototypes assume a different network architecture for within the rack(s) that form a unit of disaggregation vs. between such racks. To our knowledge, all existing DDC prototypes use specialized – even proprietary [12, 19, 24] – network technologies and protocols within a disaggregated rack(s). For example, SeaMicro uses a proprietary Torus-based topology and routing protocol within its disaggregated system; Huawei propose a PCIe-based fabric [14]; FireBox assumes an intra-FireBox network of 1Tbps Silicon photonic links interconnected by high-radix switches [31, 43]; and Intel's RSA likewise explores the use of Silicon photonic links and switches.

Rather than simply accepting the last two design choices (rack-scale disaggregation and specialized network designs), we critically explore when and why these choices are necessary. Our rationale in this is twofold. First, these are both choices that appear to be motivated not by fundamental constraints around disaggregating memory or CPU at the hardware level, but rather by the assumption that existing networking solutions cannot meet the (bandwidth/latency) requirements that disaggregation imposes on the network. To our knowledge, however, there has been no published evaluation showing this to be the case; hence, we seek to develop quantifiable arguments that either confirm or refute the need for these choices.

Second, these choices are likely to complicate or delay the deployment of DDC. The use of a different network architecture within vs. between disaggregated islands leads to the complexity of a two-tier heterogeneous network architecture with different protocols, configuration APIs, etc., for each; e.g., in the context of their FireBox system, the authors envisage the use of special gateway devices that translate between their custom intra-FireBox protocols and TCP/IP that is used between FireBox systems; Huawei's DC3.0 makes similar assumptions. Likewise, many of the specialized technologies these systems use (e.g., Si-photonic [59]) are still far from mainstream. Hence, once again, rather than assume change is necessary, we evaluate the possibility of maintaining a uniform "flat" network architecture based on existing commodity components as advocated in prior work [28, 38, 39].

2.2 Assumptions: System Architecture

In contrast to our assumptions regarding hardware which we based on existing prototypes, we have less to guide us on the systems front. We thus make the following assumptions, which we believe are reasonable:

System abstractions for *logical* resource aggregations. In a DDC, we will need system abstractions that represent a logical aggregation of resources, in terms of which we implement resource allocation and scheduling. One such abstraction in existing datacenters is a VM: operators provision VMs to aggregate slices of hardware resources within a server, and schedulers place jobs across VMs. While not strictly necessary, we note that the VM model can still be useful in DDC.[4] For convenience, in this paper we assume that computational resources are still aggregated to form VMs (or VM-like constructs), although now the resources assigned to a VM come from distributed hardware blades. Given a VM (or VM-like) abstraction, we assign resources to VMs differently based on the *scope* of disaggregation that we assume: for rack-scale disaggregation, a VM is assigned resources from within a single rack while, for datacenter-scale disaggregation, a VM is assigned resources from anywhere in the datacenter.

Hardware organization. We assume that resources are organized in racks as in today's datacenters. We assume a "mixed" organization in which each rack hosts a mix of different types of resource blades, as opposed to a "segregated" organization in which a rack is populated with a single

[4]In particular, continuing with the abstraction of a VM would allow existing software infrastructure — i.e., hypervisors, operating systems, datacenter middleware, and applications — to be reused with little or no modification.

type of resource (e.g., all memory blades). This leads to a more uniform communication pattern which should simplify network design and also permits optimizations that aim to localize communication; e.g., co-locating a VM within a rack, which would not be possible with a segregated organization.

Page-level remote memory access. In traditional servers, the typical memory access between CPU and DRAM occurs in the unit of a cache-line size (64B in x86). In contrast, we assume that CPU blades access remote memory at the granularity of a page (4KB in x86), since page-level access has been shown to better exploit spatial locality in common memory access patterns [46]. Moreover, this requires little or no modification to the virtual memory subsystems of hypervisors or operating systems, and is completely transparent to user-level applications.

Block-level distributed data placement. We assume that applications in DDC read and write large files at the granularity of "sectors" (512B in x86). Furthermore, the disk block address space is range partitioned into "blocks", that are uniformly distributed across the disk blades. The latter is partially motivated by existing distributed file systems (*e.g.*, HDFS) and also enables better load balancing.

2.3 Design knobs

Given the above assumptions, we are left with two key system design choices that we treat as "knobs" in our study: *the amount of local memory on compute blades* and *the scope of disaggregation* (e.g., rack- or datacenter-scale). We explore how varying these knobs impacts the network requirements and traffic characteristics in DDC in the following section.

The remainder of this paper is organized as follows. We first analyze network-layer bandwidth and latency requirements in DDC (§3) *without* considering contention between network flows, then in §4 relax this constraint. We end with a discussion of the future directions in §5.

3 Network Requirements

We start by evaluating network latency and bandwidth requirements for disaggregation. We describe our evaluation methodology (§3.1), present our results (§3.2) and then discuss their implications (§3.3).

3.1 Methodology

In DDC, traffic between resources that was contained within a server is now carried on the "external" network. As with other types of interconnects, the key requirement will be low latency and high throughput to enable this disaggregation. We review the forms of communication between resources within a server in Table 1 to examine the feasibility of such a network. As mentioned in §2, CPU-to-CPU cache coherence traffic does not cross the external network. For I/O traffic to storage devices, the current latency and bandwidth requirements are such that we can expect to consolidate them into the network fabric with low performance impact, assuming we have a 40Gbps or 100Gbps network. Thus, the dominant impact to application performance will come from CPU-memory disaggregation; hence, we focus on evaluating the network bandwidth and latency required to support remote memory.

As mentioned earlier, we assume that remote memory is managed at the page granularity, in conjunction with virtual memory page replacement algorithms implemented by the hypervisor or operating system. For each paging operation there are two main sources of performance penalty: i) the software overhead for trap and page eviction and ii) the time to transfer pages over the network. Given our focus on network requirements, we only consider the latter in this paper (modulo a brief discussion on current software overheads later in this section).

Applications. We use workloads from diverse applications running on real-world and benchmark datasets, as shown in Table 2. The workloads can be classified into two classes based on their performance characteristics. We elaborate briefly on our choice to take these applications as is, rather than seek to optimize them for DDC. Our focus in this paper is on understanding whether and why networking might gate the deployment of DDC. For this, we are interested in the degradation that applications might suffer if they were to run in DDC. We thus compare the performance of an application in a server-centric architecture to its performance in the disaggregated context we consider here (with its level of bandwidth and local memory). This would be strictly worse than if we compared to the application's performance if it had been rewritten for this disaggregated context. Thus, legacy (i.e., server-centric) applications represent the worst-case in terms of potential degradation and give us a lower bound on the network requirements needed for disaggregation (it might be that rewritten applications could make do with lower bandwidths). Clearly, if new networking technologies exceed this lower bound, then all applications (legacy and "native" DDC) will benefit. Similarly, new programming models designed to exploit disaggregation can only improve the performance of all applications. The question of how to achieve improved performance through new technologies and programming models is an interesting one but beyond the scope of our effort and hence one we leave to future work.

Emulating remote memory. We run the following applications unmodified with 8 threads and reduce the amount of local memory directly accessible by the applications.

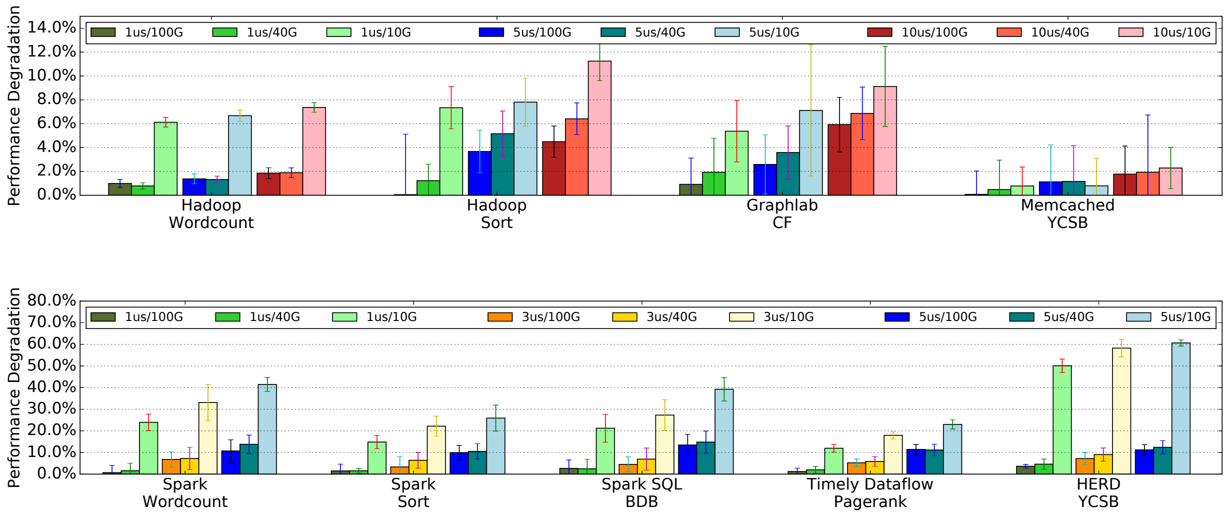


Figure 2: Comparison of application-level performance in disaggregated datacenters with respect to existing server-centric architectures for different latency/bandwidth configurations and 25% local memory on CPU blades — Class A apps (top) and Class B apps (bottom). To maintain application-level performance within reasonable performance bounds ($\sim$5% on an average), Class A apps require 5μs end-to-end latency and 40Gbps bandwidth, and Class B apps require 3μs end-to-end latency and $40-100$Gbps bandwidth. See §3.2 for detailed discussion.

To emulate remote memory accesses, we implement a special swap device backed by the remaining physical memory rather than disk. This effectively partitions main memory into "local" and "remote" portions where existing page replacement algorithms control when and how pages are transferred between the two. We tune the amount of "remote" memory by configuring the size of the swap device; remaining memory is "local". We intercept all page faults and inject artificial delays to emulate network round-trip latency and bandwidth for each paging operation. Note that when a page fault occurs, the page is not actually swapped over the network; instead, it is swapped to the remaining part of the memory on the same machine.

We measure relative application-level performance on the basis of job completion time as compared to the zero-delay case. Thus, our results do not account for the delay introduced by software overheads for page operations and should be interpreted as *relative* performance degradations over different network configurations. Note too that the delay we inject is purely an artificial parameter and hence does not (for example) realistically model queuing delays that may result from network congestion caused by the extra traffic due to disaggregation; we consider network-wide traffic and effects such as congestion in §4.

Testbed. Each application operates on $\sim$ 125GB of data equally distributed across an Amazon EC2 cluster comprising 5 `m3.2xlarge` servers. Each of these servers has 8 vCPUs, 30GB main memory, 2×80GB SSD drives and a 1Gbps access link bandwidth. We enabled EC2's Virtual Private Network (VPC [3]) capability in our cluster to ensure no interference with other Amazon EC2 instances.

We verified that `m3.2xlarge` instances' 1Gbps access links were not a bottleneck to our experiment in two ways. First, in all cases where the network approached full utilization, CPU was fully utilized, indicating that the CPU was not blocked on network calls. Next, we ran our testbed on `c3.4xlarge` instances with 2Gbps access links (increased network bandwidth with roughly the same CPU). We verified that even with more bandwidth, all applications for which link utilization was high maintained high CPU utilization. This aligns with the conclusions drawn in [53].

We run batch applications (Spark, Hadoop, Graphlab, and Timely Dataflow) in a cluster with 5 worker nodes and 1 master node; the job request is issued from the master node. For point-query applications (memcached, HERD), requests are sent from client to server across the network. All applications are multi-threaded, with the same number of threads as cores. To compensate for the performance noise on EC2, we run each experiment 10 times and take the median result.

3.2 Results

We start by evaluating application performance in a disaggregated vs. a server-centric architecture. Figure 2 plots the performance degradation for each application under

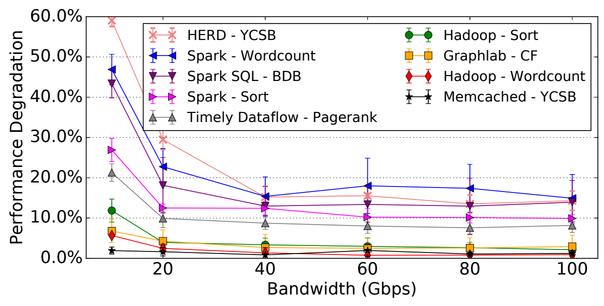


Figure 3: Impact of network bandwidth on the results of Figure 2 for end-to-end latency fixed to 5μs and local memory fixed to 25%.

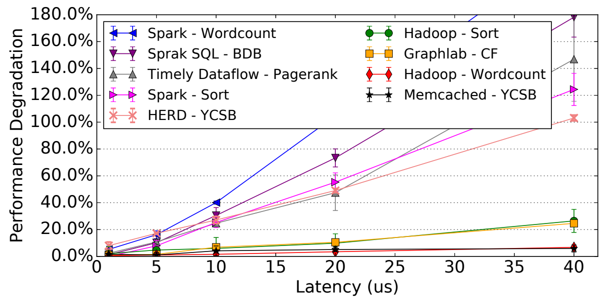


Figure 4: Impact of network latency on the results of Figure 2 for bandwidth fixed to 40Gbps and local memory fixed to 25%.

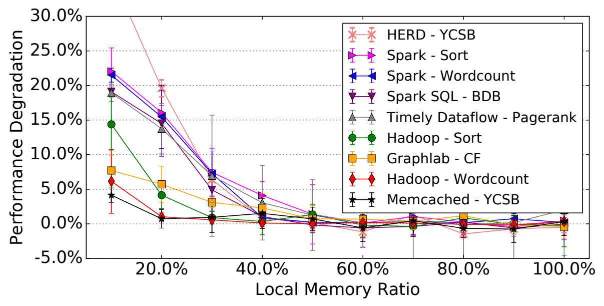


Figure 5: Impact of "local memory" on the results of Figure 2 for end-to-end latency fixed to 5μs and network bandwidth 40Gbps. Negative values are due to small variations in timings between runs.

different assumptions about the latency and bandwidth to remote memory. In these experiments, we set the local memory in the disaggregated scenario to be 25% of that in the server-centric case (we will examine our choice of 25% shortly). Note that the injected latency is constant across requests; we leave studying the effects of possibly high tail

Network Provision	**Class A**	**Class B**
5μs, 40Gbps	20%	35%
3μs, 100Gbps	15%	30%

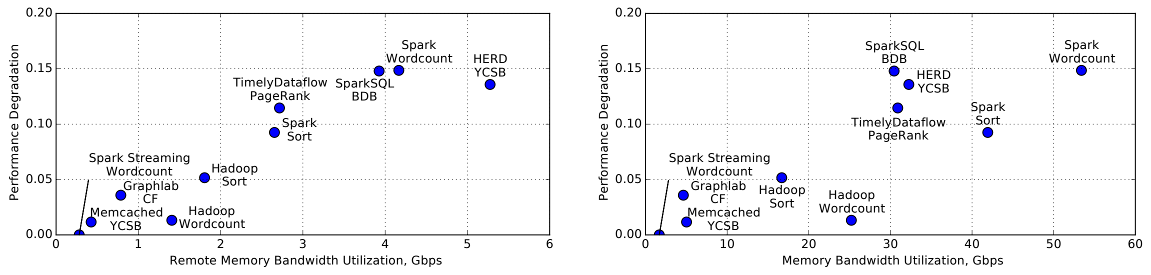
Table 3: Class B apps require slightly higher local memory than Class A apps to achieve an average performance penalty under 5% for various latency-bandwidth configurations.

latencies to future work.

From Figure 2, we see that our applications can be broadly divided into two categories based on the network latency and bandwidth needed to achieve a low performance penalty. For example, for the applications in Fig. 2 (top) — Hadoop Wordcount, Hadoop Sort, Graphlab and Memcached — a network with an end-to-end latency of 5μs and bandwidth of 40Gbps is sufficient to maintain an average performance penalty under 5%. In contrast, the applications in Fig. 2 (bottom) — Spark Wordcount, Spark Sort, Timely, SparkSQL BDB, and HERD — require network latencies of 3μs and 40 – 100Gbps bandwidth to maintain an average performance penalty under 8%. We term the former applications *Class A* and the latter *Class B* and examine the feasibility of meeting their respective requirements in §3.3. We found that Spark Streaming has a low memory utilization. As a result, its performance degradation is near zero in DDC, and we show it only in Figure 6.

Sensitivity analysis. Next, we evaluate the sensitivity of application performance to network bandwidth and latency. Fig. 3 plots the performance degradation under increasing network bandwidth assuming a fixed network latency of 5μs while Fig. 4 plots degradation under increasing latency for a fixed bandwidth of 40Gbps; in both cases, local memory is set at 25% as before. We see that beyond 40Gbps, increasing network bandwidth offers little improvement in application-level performance. In contrast, performance — particularly for Class B apps — is very sensitive to network latency; very low latencies (3 – 5μs) are needed to avoid non-trivial performance degradation.

Finally, we measure how the amount of local memory impacts application performance. Figure 5 plots the performance degradation that results as we vary the fraction of local memory from 100% (which corresponds to no CPU-memory disaggregation) down to 10%, assuming a fixed network latency and bandwidth of 5μs and 40Gbps respectively; note that the 25% values (interpolated) in Figure 5 correspond to 5μs, 40Gbps results in Figure 2. As expected, we see that Class B applications are more sensitive to the amount of local memory than Class A apps; e.g., increasing the amount of local memory from 20% to 30% roughly halves the performance degradation in Class B from approximately 15% to



(a) Remote Memory Bandwidth Utilization

(b) Memory Bandwidth Utilization

Figure 6: Performance degradation of applications is correlated with the swap memory bandwidth and overall memory bandwidth utilization.

7%. In all cases, increasing the amount of local memory beyond 40% has little to no impact on performance degradation.

Understanding (and extrapolating from) our results. One might ask *why* we see the above requirements – i.e., what characteristic of the applications we evaluated led to the specific bandwidth and latency requirements we report? An understanding of these characteristics could also allow us to generalize our findings to other applications.

We partially answer this question using Figure 6, which plots the performance degradation of the above nine workloads against their swap and memory bandwidth[5]. Figure 6(a) and 6(b) show that an application's performance degradation is very strongly correlated with its swap bandwidth and well correlated with its memory bandwidth. The clear correlation with swap bandwidth is to be expected. That the overall memory bandwidth is also well correlated with the resultant degradation is perhaps less obvious and an encouraging result as it suggests that an application's memory bandwidth requirements might serve as a rough indicator of its expected degradation under disaggregation: this is convenient as memory bandwidth is easily measured without requiring any of our instrumentation (i.e., emulating remote memory by a special swap device, etc.). Thus it should be easy for application developers to get a rough sense of the performance degradation they might expect under disaggregation and hence the urgency of rewriting their application for disaggregated contexts.

We also note that there is room for more accurate predictors: the difference between the two figures (Figs. 6(a) and 6(b)) shows that the locality in memory access patterns does play some role in the expected degradation (since the swap bandwidth which is a better predictor captures only the subset of memory accesses that miss in local memory). Building better prediction models that account for an application's memory access pattern is an interesting question that we leave to future work.

Access Granularity. Tuning the granularity of remote memory access is an interesting area for future work. For example, soNUMA [51] accesses remote memory at cache-line size granularity, which is much smaller than page-size. This may allow point-query applications to optimize their dependence on remote memory. On the other hand, developers of applications which use large, contiguous blocks of memory may wish to use hugepages to reduce the number of page table queries and thus speed up virtual memory mapping. Since Linux currently limits (non-transparent) hugepages from being swapped out of physical memory, exploring this design option is not currently feasible.

Overall, we anticipate that programmers in DDC will face a tradeoff in optimizing their applications for disaggregation depending on its memory access patterns.

Remote SSD and NVM. Our methodology is not limited to swapping to remote memory. In fact, as long as the 3μs latency target is met, there is no limitation on the media of the remote storage. We envision that the remote memory could be replaced by SSD or forthcoming Non-Volatile Memory (NVM) technologies, and anticipate different price and performance tradeoff for these technologies.

Summary of results. In summary, supporting memory disaggregation while maintaining application-level performance within reasonable bounds imposes certain requirements on the network in terms of the end-to-end latency and bandwidth it must provide. Moreover, these requirements

[5]We use Intel's Performance Counter Monitor software [18] to read the uncore performance counters that measure the number of bytes written to and read from the integrated memory controller on each CPU. We confirmed using benchmarks designed to saturate memory bandwidth [4] that we could observe memory bandwidth utilization numbers approaching the reported theoretical maximum. As further validation, we verified that our Spark SQL measurement is consistent with prior work [55].

are closely related to the amount of local memory available to CPU blades. Table 3 summarizes these requirements for the applications we studied. We specifically investigate a few combinations of network latency, bandwidth, and the amount of local memory needed to maintain a performance degradation under 5%. We highlight these design points because they represent what we consider to be sweet spots in achievable targets both for the amount of local memory and for network requirements, as we discuss next.

3.3 Implications and Feasibility

We now examine the feasibility of meeting the requirements identified above.

Local memory. We start with the requirement of between 20 – 30% local memory. In our experiments, this corresponds to between 1.50−2.25GB/core. We look to existing hardware prototypes for validation of this requirement. The FireBox prototype targets 128GB of local memory shared by 100 cores leading to 1.28GB/core,[6] while the analysis in [46] uses 1.5GB/core. Further, [47] also indicates 25% local memory as a desirable setting, and HP's "The Machine" [2] uses an even larger fraction of local memory: 87%. Thus we conclude that our requirement on local memory is compatible with demonstrated hardware prototypes. Next, we examine the feasibility of meeting our targets for network bandwidth and latency.

Network bandwidth. Our bandwidth requirements are easily met: 40Gbps is available today in commodity datacenter switches *and* server NICs [16]; in fact, even 100Gbps switches and NICs are available, though not as widely [1]. Thus, ignoring the potential effects of congestion (which we consider next in §4), providing the network bandwidth needed for disaggregation should pose no problem. Moreover, this should continue to be the case in the future because the trend in link bandwidths currently exceeds that in number of cores [5, 7, 11].

Network latency. The picture is less clear with respect to latency. In what follows, we consider the various components of network latency and whether they can be accommodated in our target budget of 3μs (for Class B apps) to 5μs (for Class A apps).

Table 4 lists the six components of the end-to-end latency incurred when fetching a 4KB page using 40Gbps links, together with our estimates for each. Our estimates are based on the following common assumptions about existing datacenter networks: (1) the one-way path between servers in different racks crosses three switches (two ToR and one fabric switch) while that between servers in the same rack crosses a single ToR switch, (2) inter-rack distances of 40m and intra-rack distances of 4m with a propagation speed of 5ns/m, (3) cut-through switches.[7] With this, our round-trip latency includes the software overheads associated with moving the page to/from the NIC at both the sending and receiving endpoints (hence 2x the OS and data copy overheads), 6 switch traversals, 4 link traversals in each direction including two intra-rack and two cross-rack, and the transmission time for a 4KB page (we ignore transmission time for the page request), leading to the estimates in Table 4.

We start by observing that the network introduces three unavoidable latency overheads: (i) the data transmission time, (ii) the propagation delay; and (iii) the switching delay. Together, these components contribute to roughly 3.14μs across racks and 1.38μs within a rack.[8]

In contrast, the network software at the endpoints is a significant contributor to the end-to-end latency! Recent work reports a round-trip kernel processing time of 950 ns measured on a 2.93GHz Intel CPU running FreeBSD (see [56] for details), while [52] reports an overhead of around 1μs to copy data between memory and the NIC. With these estimates, the network software contributes roughly 3.9μs latency — this represents 55% of the end-to-end latency in our baseline inter-rack scenario and 73% in our baseline intra-rack scenario.

The end-to-end latencies we estimated in our baseline scenarios (whether inter- or intra-rack) fail to meet our target latencies for either Class B or Class A applications. Hence, we consider potential optimizations and technologies that can reduce these latencies. Two technologies show promise: RDMA and integrated NICs.

Using RDMA. RDMA effectively bypasses the packet processing in the kernel, thus eliminating the OS overheads from Table 4. Thus, using RDMA (Infiniband [15] or Omnipath [17]), we estimate a reduced end-to-end latency of 5.14μs across racks (column #4 in Table 4) and 3.38μs within a rack.

Using NIC integration. Recent industry efforts pursue the integration of NIC functions closer to the CPU [33] which would reduce the overheads associated with copying data to/from the NIC. Rosenblum *et al.* [57] estimate that such integration together with certain software optimizations can reduce copy overheads to sub-microseconds, which we estimate at 0.5μs (similar to [57]).

[6] We thank Krste Asanović for clarification on FireBox's technical specs.

[7] As before, we ignore the queuing delays that may result from congestion at switches – we will account for this in §4.

[8] Discussions with switch vendors revealed that they are approaching the fundamental limits in reducing switching delays (for electronic switches), hence we treat the switching delay as unavoidable.

Component	Baseline (μs)		With RDMA (μs)		With RDMA + NIC Integr. (μs)	
	Inter-rack	**Intra-rack**	**Inter-rack**	**Intra-rack**	**Inter-rack**	**Intra-rack**
OS	2×0.95	2×0.95	0	0	0	0
Data copy	2×1.00	2×1.00	2×1.00	2×1.00	2×0.50	2×0.50
Switching	6×0.24	2×0.24	6×0.24	2×0.24	6×0.24	2×0.24
Propagation (Inter-rack)	4×0.20	0	4×0.20	0	4×0.20	0
Propagation (Intra-rack)	4×0.02	4×0.02	4×0.02	4×0.02	4×0.02	4×0.02
Transmission	1×0.82	1×0.82	1×0.82	1×0.82	1×0.82	1×0.82
Total	7.04μ**s**	5.28μs	5.14μs	3.38μs	4.14μ**s**	2.38μ**s**

Table 4: Achievable round-trip latency (Total) and the components that contribute to the round-trip latency (see discussion in §3.3) on a network with 40Gbps access link bandwidth (one can further reduce the **Total** by 0.5μs using 100Gbps access link bandwidth). The baseline denotes the latency achievable with existing network technology. The fractional part in each cell is the latency for one traversal of the corresponding component and the integral part is the number of traversal performed in one round-trip time (see discussion in §3.3).

Using RDMA and NIC integration. As shown in column #5 in Table 4, the use of RDMA together with NIC integration reduces the end-to-end latency to 4.14μs across racks; within a rack, this further reduces down to 2.38μs (using the same differences as in column #2 and column #3).

Takeaways. We highlight a few takeaways from our analysis:

- The overhead of network *software* is the key barrier to realizing disaggregation with current networking technologies. Technologies such as RDMA and integrated NICs that eliminate some of these overheads offer promise: reducing end-to-end latencies to 4.14μs between racks and 2.38μs within a rack. However, demonstrating such latencies in a working prototype remains an important topic for future exploration.
- Even assuming RDMA and NIC integration, the end-to-end latency across racks (4.14μs) meets our target latency only for Class A, but not Class B, applications. Our target latency for Class B apps is only met by the end-to-end latency within a rack. Thus, Class B jobs will have to be scheduled within a single rack (or nearby racks). That is, while Class A jobs can be scheduled at blades distributed across the datacenter, Class B jobs will need to be scheduled within a rack. The design and evaluation of such schedulers remains an open topic for future research.
- While new network hardware such as high-bandwidth links (e.g., 100Gbps or even 1Tbps as in [31, 43]) and high-radix switches (*e.g.*, 1000 radix switch [31]) are certainly useful, they optimize a relatively small piece of the overall latency in our baseline scenario technologies. All-optical switches also fall into this category – providing both potentially negligible switching delay and high bandwidth. That said, once we assume the benefits of RDMA and NIC integration, then the contribution of new links and switches could bring even the cross-rack latency to within our 3μs target for Class B applications, enabling true datacenter-scale disaggregation; e.g., using 100Gbps links reduces the end-to-end latency to 3.59μs between racks, extremely close to our 3μs.
- Finally, we note that managing network congestion to achieve zero or close-to-zero queuing within the network will be essential; e.g., a packet that is delayed such that it is queued behind (say) 4 packets will accumulate an additional delay of $4 \times 0.82\mu$s! Indeed, reducing such transmission delays may be the reason to adopt high-speed links. We evaluate the impact of network congestion in the following section.

4 Network Designs for Disaggregation

Our evaluation has so far ignored the impact of queuing delay on end-to-end latency and hence application performance; we remedy the omission in this section. The challenge is that queuing delay is a function of the overall network design, including: the traffic workload, network topology and routing, and the end-to-end transport protocol. Our evaluation focuses on existing proposals for transport protocols, with standard assumptions about the datacenter topology and routing. However, the input traffic workload in DDC will be very different from that in a server-centric datacenter and, to our knowledge, no models exist that characterize traffic in a DDC.

We thus start by devising a methodology that extends our experimental setup to generate an application-driven input traffic workload (§4.1), then describe how we use this traffic model to evaluate the impact of queuing delay (§4.2). Finally, we present our results on: (i) how existing transport designs perform under DDC traffic workloads (§4.3), and (ii) how existing transport designs impact end-to-end application performance (§4.4). To our knowledge, our results represent

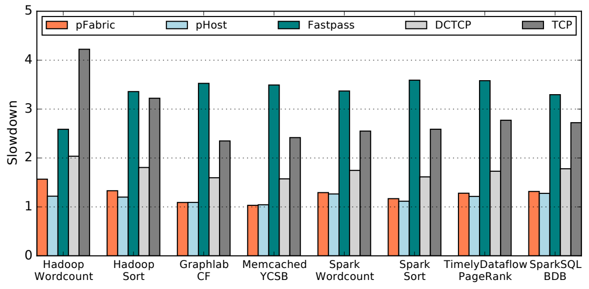


Figure 7: The performance of the five protocols for the case of 100Gbps access link capacity. The results for 40Gbps access links lead to similar conclusions. See §4.3 for discussion on these results.

the first evaluation of transport protocols for DDC.

4.1 Methodology: DDC Traffic Workloads

Using our experimental setup from §3.1, we collect a remote memory access trace from our instrumentation tool as described in §3.1, a network access trace using `tcpdump` [25], and a disk access trace using `blktrace`.

We translate the accesses from the above traces to network flows in our simulated disaggregated cluster by splitting each node into one compute, one memory, and one disk blade and assigning memory blades to virtual nodes.

All memory and disk accesses captured above are associated with a specific address in the corresponding CPU's global virtual address space. We assume this address space is uniformly partitioned across all memory and disk blades reflecting our assumption of distributed data placement (§2.2).

One subtlety remains. Consider the disk accesses at a server *A* in the original cluster: one might view all these disk accesses as corresponding to a flow between the compute and disk blades corresponding to *A*, but in reality *A*'s CPU may have issued some of these disk accesses in response to a request from a remote server *B* (*e.g.*, due to a shuffle request). In the disaggregated cluster, this access should be treated as a network flow between *B*'s compute blade and *A*'s disk blade.

To correctly attribute accesses to the CPU that originates the request, we match network and disk traces across the cluster – e.g., matching the network traffic between *B* and *A* to the disk traffic at *A* – using a heuristic based on both the timestamps and volume of data transferred. If a locally captured memory or disk access request matches a local flow in our `tcpdump` traces, then it is assumed to be part of a remote read and is attributed to the remote endpoint of the network flow. Otherwise, the memory/disk access is assumed to have originated from the local CPU.

4.2 Methodology: Queuing delay

We evaluate the use of existing network designs for DDC in two steps. First, we evaluate how existing network designs fare under DDC traffic workloads. For this, we consider a suite of state-of-the-art network designs and use simulation to evaluate their network-layer performance – measured in terms of flow completion time (FCT) – under the traffic workloads we generate as above. We then return to actual execution of our applications (Table 2) and once again emulate disaggregation by injecting latencies for page misses. However, now we inject the flow completion times obtained from our best-performing network design (as opposed to the constant latencies from §3). This last step effectively "closes the loop", allowing us to evaluate the impact of disaggregation on application-level performance for realistic network designs and conditions.

Simulation Setup. We use the same simulation setup as prior work on datacenter transports [29, 30, 36]. We simulate a topology with 9 racks (with 144 total endpoints) and a full bisection bandwidth Clos topology with 36KB buffers per port; our two changes from prior work are to use 40Gbps or 100Gbps access links (as per §3), and setting propagation and switching delays as discussed in §3.3 (Table 4 with RDMA and NIC integration). We map the 5 EC2-node cluster into a disaggregated cluster with 15 blades: 5 each of compute, memory and disk. Then, we extract the flow size and inter-arrival time distribution for each endpoint pair in the 15 blades disaggregated cluster, and generate traffic using the distributions. Finally, we embed the multiple disaggregated clusters into the 144-endpoint datacenter with both rack-scale and datacenter-scale disaggregation, where communicating nodes are constrained to be within a rack and unconstrained, respectively.

We evaluate five protocols; in each case, we set protocol-specific parameters following the default settings but adapted to our bandwidth-delay product as recommended.

1. **TCP**, with an initial congestion window of 2.
2. **DCTCP**, which leverages ECN for enhanced performance in datacenter contexts.
3. **pFabric**, approximates shortest-job-first scheduling in a network context using switch support to prioritize flows with a smaller remaining flow size [30]. We set pFabric to have an initial congestion window of 12 packets and a retransmission timeout of 45μs.
4. **pHost**, emulates pFabric's behavior but using only scheduling at the end hosts [36] and hence allows the use of commodity switches. We set pHost to have a free token limit of 8 packets and a retransmission timeout of 9.5μs as recommended in [36].

5. **Fastpass**, introduces a centralized scheduler that schedules every packet. We implement Fastpass's [54] scheduling algorithm in our simulator as described in [36] and optimistically assume that the scheduler's decision logic itself incurs no overhead (i.e., takes zero time) and hence we only consider the latency and bandwidth overhead of contacting the central scheduler. We set the Fastpass epoch size to be 8 packets.

4.3 Network-level performance

We evaluate the performance of our candidate transport protocols in terms of their mean slowdown [30], which is computed as follows. The slowdown for a flow is computed by dividing the flow completion time achieved in simulation by the time that the flow would take to complete if it were alone in the network. The mean slowdown is then computed by averaging the slowdown over all flows. Figure 7 plots the mean slowdown for our five candidate protocols, using 100Gbps links (all other parameters are as in §4.2).

Results. We make the following observations. First, while the relative ordering in mean slowdown for the different protocols is consistent with prior results [36], their *absolute* values are higher than reported in their original papers; e.g. pFabric and pHost both report close-to-optimal slowdowns with values close to 1.0 [30,36]. On closer examination, we found that the higher slowdowns with disaggregation are a consequence of the differences in our traffic workloads (both earlier studies used heavy-tailed traffic workloads based on measurement studies from existing datacenters). In our DDC workload, reflecting the application-driven nature of our workload, we observe many flow arrivals that appear very close in time (only observable on sub-10s of microsecond timescales), leading to high slowdowns for these flows. This effect is strongest in the case of the Wordcount application, which is why it suffers the highest slowdowns. We observed similar results in our simulation of rack-scale disaggregation (graph omitted).

4.4 Application-level performance

We now use the pFabric FCTs obtained from the above simulations as the memory access times in our emulation methodology from §3.

We measure the degradation in application performance that results from injecting remote memory access times drawn from the FCTs that pFabric achieves with 40Gbps links and with 100Gbps links, in each case considering both datacenter-wide and rack-scale disaggregation. As in §3, we measure performance degradation compared to the baseline of performance without disaggregation (i.e., injecting zero latency).

In all cases, we find that the inclusion of queuing delay

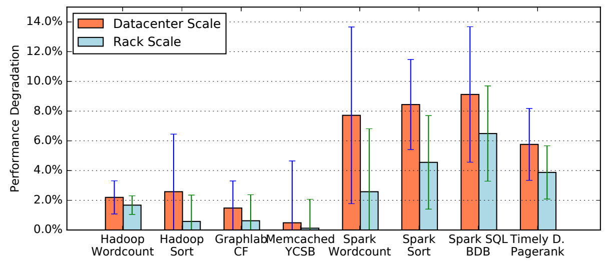


Figure 8: Application layer slowdown for each of the four applications at rack-scale and datacenter scale after injecting pFabric's FCT with 100Gbps link.

does have a non-trivial impact on performance degradation at 40 Gbps – typically increasing the performance degradation relative to the case of zero-queuing delay by between 2-3x, with an average performance degradation of 14% with datacenter-scale disaggregation and 11% with rack-scale disaggregation.

With 100Gbps links, we see (in Figure 8) that the performance degradation ranges between 1-8.5% on average with datacenter scale disaggregation, and containment to a rack lowers the degradation to between 0.4-3.5% on average. This leads us to conclude that 100Gbps links are both required and sufficient to contain the performance impact of queuing delay.

5 Future Directions

So far, we used emulation and simulation to evaluate the minimum network requirements for disaggregation. This opens two directions for future work: (1) demonstrating an end-to-end system implementation of remote memory access that meets our latency targets, and (2) investigating programming models that actively exploit disaggregation to *improve* performance. We present early results investigating the above with the intent of demonstrating the potential for realizing positive results to the above questions: each topic merits an in-depth exploration that is out of scope for this paper.

5.1 Implementing remote memory access

We previously identified an end-to-end latency target of 3-5μs for DDC that we argued could be met with RDMA. The (promising) RDMA latencies in §4 are as reported by native RDMA-based applications. We were curious about the feasibility of realizing these latencies if we were to retain our architecture from the previous section in which remote memory is accessed as a special swap device as this would provide a simple and transparent approach to utilizing remote memory.

We thus built a kernel space RDMA block device driver which serves as a swap device; i.e., the local CPU can now

Min	Avg	Median	99.5 Pcntl	Max
3394	3492	3438	4549	12254

Table 5: RDMA block device request latency(ns)

swap to remote memory instead of disk. We implemented the block device driver on a machine with a 3 GHz CPU and a Mellanox 4xFDR Infiniband card providing 56 Gbps bandwidth. We test the block device throughput using dd with direct IO, and measure the request latency by instrumenting the driver code. The end-to-end latency of our approach includes the RDMA request latency and the latency introduced by the kernel swap itself. We focus on each in turn.

RDMA request latency. A few optimizations were necessary to improve RDMA performance in our context. First, we *batch* block requests sent to the RDMA NIC and the driver waits for all the requests to return before notifying the upper layer: this gave a block device throughput of only 0.8GB/s and latency around 4-16us. Next, we *merge* requests with contiguous addresses into a single large request: this improved throughput to 2.6GB/s (a 3*x* improvement). Finally, we allow *asynchronous* RDMA requests: we created a data structure to keep track of outgoing requests and notify the upper layer immediately for each completed request; this improves throughput to 3.3GB/s which is as high as a local RamFS, and reduces the request latency to 3-4us (Table 5). This latency is within 2x of latencies reported by native RDMA applications which we view as encouraging given the simplicity of the design and that additional optimizations are likely possible.

Swap latency. We calculated the software overhead of swapping on a commodity desktop running Linux 3.13 by simultaneously measuring the times spent in the page fault handler and accessing disk. We found that convenient measurement tools such as `ftrace` and `printk` introduce unacceptable overhead for our purposes. Thus, we wrap both the body of the `__do_page_fault` function and the call to the `swapin_readahead` function (which performs a swap from disk) in `ktime_get` calls. We then pack the result of the measurement for the `swapin_readahead` function into the unused upper 16-bits of the return value of its caller, `do_swap_page`, which propagates the value up to `__do_page_fault`.

Once we have measured the body of `__do_page_fault`, we record both the latency of the whole `__do_page_fault` routine (25.47μs), as well as the time spent in `swapin_readahead` (23.01μs). We subtract these and average to find that the software overhead of swapping is 2.46μs. This number is a lower-bound on the software overhead of the handler, because we assume that all of `swapin_readahead` is a "disk access".

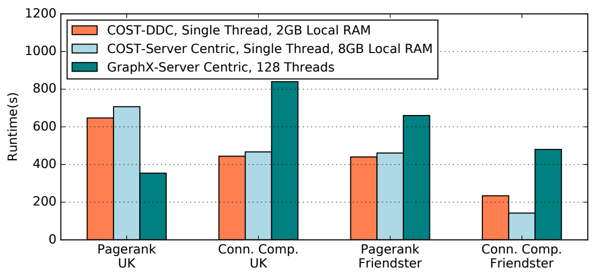


Figure 9: Running COST in a simulated DDC. COST-DDC is 1.48 to 2.05 faster than GraphX-Server Centric except for one case. We use two datasets in our evaluation, UK-2007-05 (105m nodes, 3.7b edges), and Friendster (65m nodes, 1.8b edges)

In combination with the above RDMA latencies, these early numbers suggest that a simple system design for low-latency access to remote memory could be realized.

5.2 Improved performance via disaggregation

In the longer term, one might expect to re-architect applications to actively exploit disaggregation for improved performance. One promising direction is for applications to exploit the availability of low-latency access to large pools of remote memory [46]. One approach to doing so is based on extending the line of argument in the COST work [48] by using remote memory to avoid parallelization overheads. COST is a single machine graph engine that outperforms distributed graph engines like GraphX when the graph fits into main memory. The RDMA swap device enables COST to use "infinite" remote memory when the graph is too large. We estimate the potential benefits of this approach with the following experiment. First, to model an application running in a DDC, we set up a virtual machine with 4 cores, 2GB of local memory, and access to an "infinitely" large remote memory pool by swapping to an RDMA-backed block device. Next, we consider two scenarios that represent server-centric architecture. One is a server with 4 cores and 8GB of local memory (25% larger than the DDC case as in previous sections) and an "infinitely" large local SSD swap – this represents the COST baseline in a server-centric context. Second, we evaluate GraphX using a 16-node `m2.4xlarge` cluster on EC2 – this represents the scale-out approach in current server-centric architecture. We run Pagerank and Connected Components using COST, a single-thread graph compute engine over three large graph datasets. COST `mmaps` the input file, so we store the input files on another RDMA-backed block device. Figure 9 shows the application runtime of COST-DDC, COST-SSD and GraphX-Server Centric. In all but one case, COST-DDC is 1.48 to 2.05 times faster than the GraphX (server-centric) scenario and slightly better than the server-centric COST scenario (the improvement over the latter grows with increasing data set size). Performance is worse for Pagerank on the UK-2007-5 dataset, consistent with the results in [48] because the graph in this case is more easily partitioned.

Finally, another promising direction for improving performance is through better resource utilization. As argued in [40, 46], CPU-to-memory utilization for tasks in today's datacenters varies by three orders of magnitude across tasks; by "bin packing" on a much larger scale, DDC should achieve more efficient statistical multiplexing, and hence higher resource utilization and improved job completion times. We leave an exploration of this direction to future work.

6 Related Work and Discussion

As mentioned earlier, there are many recent and ongoing efforts to prototype disaggregated hardware. We discussed the salient features of these efforts inline throughout this paper and hence we only briefly elaborate on them here.

Lim et al. [46, 47] discuss the trend of growing peak compute-to-memory ratio, warning of the "memory capacity wall" and prototype a disaggregated memory blade. Their results demonstrate that memory disaggregation is feasible and can even provide a 10x performance improvement in memory constrained environments.

Sudan et al. [58] use an ASIC based interconnect fabric to build a virtualized I/O system for better resource sharing. However, these interconnects are designed for their specific context; the authors neither discuss network support for disaggregation more broadly nor consider the possibility of leveraging known datacenter network technologies to enable disaggregation.

FireBox [31] proposes a holistic architecture redesign of datacenter racks to include 1Tbps silicon photonic links, high-radix switches, remote nonvolatile memory, and System-on-Chips (SoCs). Theia [61] proposes a new network topology that interconnects SoCs at high density. Huawei's DC3.0 (NUWA) system uses a proprietary PCIe-based interconnect. R2C2 [34] proposes new topologies, routing and congestion control designs for rack-scale disaggregation. None of these efforts evaluate network requirements based on existing workloads as we do, nor do they evaluate the effectiveness of existing network designs in supporting disaggregation or the possibility of disaggregating at scale.

In an early position paper, Han et al. [40] measure – as we do – the impact of remote memory access latency on application-level performance within a single machine. Our work extends this understanding to a larger set of workloads and concludes with more stringent requirements on latency and bandwidth than Han et al. do, due to our consideration of Class B applications. In addition, we use simulation and emulation to study the impact of queueing delay and transport designs which further raises the bar on our target network performance.

Multiple recent efforts [35, 42, 45, 52] aim to reduce the latency in networked applications through techniques that bypass the kernel networking stack, and so forth. Similarly, efforts toward NIC integration by CPU architects [33] promise to enable even further latency-saving optimizations. As we note in §3.3, such efforts are crucial enablers in meeting our latency targets.

Distributed Shared Memory (DSM) [32, 44, 50] systems create a shared address space and allow remote memory to be accessed among different endpoints. While this is a simple programming abstraction, DSM incurs high synchronization overhead. Our work simplifies the design by using remote memory only for paging, which removes synchronization between the endpoints.

Based on our knowledge of existing designs and prototypes [12, 13, 31, 46, 47], we assume partial memory disaggregation and limit the cache coherence domain to one CPU. However, future designs may relax these assumptions, causing more remote memory access traffic and cache coherence traffic. In these designs, specialized network hardware may become necessary.

7 Conclusion

This paper is a preliminary study; we have identified numerous directions for future work before disaggregation is deployable. Most important among these are the adoption of low-latency network software and hardware at endpoints, the design and implementation of a "disaggregation-aware" scheduler, and the creation of new programming models which exploit a disaggregated architecture. We believe that quantified, workload-driven studies such as that presented in this paper can serve to inform these ongoing and future efforts to build DDC systems.

Acknowledgements

We thank our shepherd Orran Krieger and the anonymous reviewers for their excellent feedback. We thank Krste Asanović for clarification on FireBox's technical specs. We thank Kostadin Ilov for his technical support on our experiments. This work is supported by Intel, NSF Grant 1420064 and Grant 1216073.

References

[1] 100G CLR4 White Paper. http://www.intel.com/content/www/us/en/research/intel-labs-clr4-white-paper.html.

[2] A look at The Machine. https://lwn.net/Articles/655437/.

[3] Amazon VPC. https://aws.amazon.com/vpc/.

[4] Bandwidth: a memory bandwidth benchmark. http://zsmith.co/bandwidth.html.

[5] Bandwidth Growth and The Next Speed of Ethernet. http://goo.gl/C5lovt.

[6] Berkeley Big Data Benchmark. https://amplab.cs.berkeley.edu/benchmark/.

[7] Big Data System research: Trends and Challenges. http://goo.gl/38qr1O.

[8] Facebook Disaggregated Rack. http://goo.gl/6h2Ut.

[9] Friendster Social Network. https://snap.stanford.edu/data/com-Friendster.html.

[10] Graphics Processing Unit. http://www.nvidia.com/object/what-is-gpu-computing.html.

[11] Here's How Many Cores Intel Corporation's Future 14-Nanometer Server Processors Will Have. http://goo.gl/y2nWOR.

[12] High Throughput Computing Data Center Architecture. http://www.huawei.com/ilink/en/download/HW_349607.

[13] HP The Machine. http://www.hpl.hp.com/research/systems-research/themachine/.

[14] Huawei NUWA. http://nuwabox.com.

[15] InfiniBand. http://www.infinibandta.org/content/pages.php?pg=about_us_infiniband.

[16] Intel Ethernet Converged Network Adapter XL710 10/40 GbE. http://www.intel.com/content/www/us/en/network-adapters/converged-network-adapters/ethernet-xl710-brief.html.

[17] Intel Omnipath. http://www.intel.com/content/www/us/en/high-performance-computing-fabrics/omni-path-architecture-fabric-overview.html.

[18] Intel Performance Counter Monitor. https://software.intel.com/en-us/articles/intel-performance-counter-monitor.

[19] Intel RSA. http://www.intel.com/content/www/us/en/architecture-and-technology/rsa-demo-x264.html.

[20] Memcached - A Distributed Memory Object Caching System. http://memcached.org.

[21] Memristor. http://www.memristor.org/reference/research/13/what-are-memristors.

[22] Netflix Rating Trace. http://www.select.cs.cmu.edu/code/graphlab/datasets/.

[23] Non-Volatile Random Access Memory. https://en.wikipedia.org/wiki/Non-volatile_random-access_memory.

[24] SeaMicro Technology Overview. http://seamicro.com/sites/default/files/SM_TO01_64_v2.5.pdf.

[25] "tcpdump". http://www.tcpdump.org.

[26] Timely Dataflow. https://github.com/frankmcsherry/timely-dataflow.

[27] Wikipedia Dump. https://dumps.wikimedia.org/.

[28] M. Al-Fares, A. Loukissas, and A. Vahdat. A Scalable, Commodity Data Center Network Architecture. SIGCOMM 2008.

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

[30] M. Alizadeh, S. Yang, M. Sharif, S. Katti, N. McKeown, B. Prabhakar, and S. Shenker. pFabric: Minimal Near-optimal Datacenter Transport. SIGCOMM 2013.

[31] K. Asanović. FireBox: A Hardware Building Block for 2020 Warehouse-Scale Computers. FAST 2014.

[32] J. K. Bennett, J. B. Carter, and W. Zwaenepoel. Munin: Distributed shared memory based on type-specific memory coherence. PPOPP 1990.

[33] N. L. Binkert, A. G. Saidi, and S. K. Reinhardt. Integrated Network Interfaces for High-bandwidth TCP/IP. ASPLOS 2006.

[34] P. Costa, H. Ballani, K. Razavi, and I. Kash. R2C2: A Network Stack for Rack-scale Computers. SIGCOMM 2015.

[35] A. Dragojević, D. Narayanan, O. Hodson, and M. Castro. FaRM: Fast Remote Memory. NSDI 2014.

[36] P. X. Gao, A. Narayan, G. Kumar, R. Agarwal, S. Ratnasamy, and S. Shenker. pHost: Distributed Near-optimal Datacenter Transport Over Commodity Network Fabric. CoNEXT 2015.

[37] A. Greenberg. SDN for the Cloud. SIGCOMM 2015.

[38] A. Greenberg, J. Hamilton, D. Maltz, and P. Patel. The Cost of a Cloud: Research Problems in Data Center Networks. ACM SIGCOMM CCR 2009.

[39] 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. SIGCOMM 2009.

[40] S. Han, N. Egi, A. Panda, S. Ratnasamy, G. Shi, and S. Shenker. Network Support for Resource Disaggregation in Next-generation Datacenters. HotNets 2013.

[41] Intel LAN Access Division. An Introduction to SR-IOV Technology. `http://goo.gl/m7jP3`.

[42] A. Kalia, M. Kaminsky, and D. G. Andersen. Using RDMA Efficiently for Key-Value Services. SIGCOMM 2014.

[43] S. Kumar. Petabit Switch Fabric Design. Master's thesis, EECS Department, University of California, Berkeley, 2015.

[44] K. Li and P. Hudak. Memory coherence in shared virtual memory systems. TOCS 1989.

[45] H. Lim, D. Han, D. G. Andersen, and M. Kaminsky. MICA: A Holistic Approach to Fast In-memory Key-Value Storage. NSDI 2014.

[46] K. Lim, J. Chang, T. Mudge, P. Ranganathan, S. K. Reinhardt, and T. F. Wenisch. Disaggregated Memory for Expansion and Sharing in Blade Servers. ISCA 2009.

[47] K. Lim, Y. Turner, J. R. Santos, A. Auyoung, J. Chang, P. Ranganathan, and T. F. Wenisch. System-level Implications of Disaggregated Memory. HPCA 2012.

[48] F. McSherry, M. Isard, and D. G. Murray. Scalability! But at What Cost? HotOS 2015.

[49] D. G. Murray, F. McSherry, R. Isaacs, M. Isard, P. Barham, and M. Abadi. Naiad: A Timely Dataflow System. SOSP 2013.

[50] J. Nelson, B. Holt, B. Myers, P. Briggs, L. Ceze, S. Kahan, and M. Oskin. Latency-tolerant Software Distributed Shared Memory. USENIX ATC 2015.

[51] S. Novakovic, A. Daglis, E. Bugnion, B. Falsafi, and B. Grot. Scale-out NUMA. ASPLOS 2014.

[52] J. Ousterhout, A. Gopalan, A. Gupta, A. Kejriwal, C. Lee, B. Montazeri, D. Ongaro, S. J. Park, H. Qin, M. Rosenblum, S. Rumble, R. Stutsman, and S. Yang. The RAMCloud Storage System. TOCS 2015.

[53] K. Ousterhout, R. Rasti, S. Ratnasamy, S. Shenker, and B.-G. Chun. Making sense of performance in data analytics frameworks. NSDI 2015.

[54] J. Perry, A. Ousterhout, H. Balakrishnan, D. Shah, and H. Fugal. Fastpass: A Centralized "Zero-Queue" Datacenter Network. SIGCOMM 2014.

[55] P. S. Rao and G. Porter. Is Memory Disaggregation Feasible?: A Case Study with Spark SQL. ANCS 2016.

[56] L. Rizzo. netmap: A Novel Framework for Fast Packet I/O. USENIX ATC 2012.

[57] S. M. Rumble, D. Ongaro, R. Stutsman, M. Rosenblum, and J. K. Ousterhout. Its Time for Low Latency. HotOS 2011.

[58] K. Sudan, S. Balakrishnan, S. Lie, M. Xu, D. Mallick, G. Lauterbach, and R. Balasubramonian. A Novel System Architecture for Web Scale Applications Using Lightweight CPUs and Virtualized I/O. HPCA 2013.

[59] C. Sun, M. T. Wade, Y. Lee, J. S. Orcutt, L. Alloatti, M. S. Georgas, A. S. Waterman, J. M. Shainline, R. R. Avizienis, S. Lin, et al. Single-chip Microprocessor that Communicates Directly Using Light. *Nature 2015.*

[60] G. Vasiliadis, M. Polychronakis, S. Antonatos, E. P. Markatos, and S. Ioannidis. Regular Expression Matching on Graphics Hardware for Intrusion Detection. RAID 2009.

[61] M. Walraed-Sullivan, J. Padhye, and D. A. Maltz. Theia: Simple and Cheap Networking for Ultra-Dense Data Centers. HotNets-XIII.

[62] W. A. Wulf and S. A. McKee. Hitting the Memory Wall: Implications of the Obvious. *SIGARCH Comput. Archit. News, March 1995.*

TensorFlow: A system for large-scale machine learning

Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, Manjunath Kudlur, Josh Levenberg, Rajat Monga, Sherry Moore, Derek G. Murray, Benoit Steiner, Paul Tucker, Vijay Vasudevan, Pete Warden, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng

Google Brain

Abstract

TensorFlow is a machine learning system that operates at large scale and in heterogeneous environments. TensorFlow uses dataflow graphs to represent computation, shared state, and the operations that mutate that state. It maps the nodes of a dataflow graph across many machines in a cluster, and within a machine across multiple computational devices, including multicore CPUs, general-purpose GPUs, and custom-designed ASICs known as Tensor Processing Units (TPUs). This architecture gives flexibility to the application developer: whereas in previous "parameter server" designs the management of shared state is built into the system, TensorFlow enables developers to experiment with novel optimizations and training algorithms. TensorFlow supports a variety of applications, with a focus on training and inference on deep neural networks. Several Google services use TensorFlow in production, we have released it as an open-source project, and it has become widely used for machine learning research. In this paper, we describe the TensorFlow dataflow model and demonstrate the compelling performance that TensorFlow achieves for several real-world applications.

1 Introduction

In recent years, machine learning has driven advances in many different fields [3, 5, 24, 25, 29, 31, 42, 47, 50, 52, 57, 67, 68, 72, 76]. We attribute this success to the invention of more sophisticated machine learning models [44, 54], the availability of large datasets for tackling problems in these fields [9, 64], and the development of software platforms that enable the easy use of large amounts of computational resources for training such models on these large datasets [14, 20].

We have developed the TensorFlow system for experimenting with new models, training them on large datasets, and moving them into production. We have based TensorFlow on many years of experience with our first-generation system, DistBelief [20], both simplifying and generalizing it to enable researchers to explore a wider variety of ideas with relative ease. TensorFlow supports both large-scale training and inference: it efficiently uses hundreds of powerful (GPU-enabled) servers for fast training, and it runs trained models for inference in production on various platforms, ranging from large distributed clusters in a datacenter, down to running locally on mobile devices. At the same time, it is flexible enough to support experimentation and research into new machine learning models and system-level optimizations.

TensorFlow uses a unified dataflow graph to represent both the computation in an algorithm *and* the state on which the algorithm operates. We draw inspiration from the high-level programming models of dataflow systems [2, 21, 34] and the low-level efficiency of *parameter servers* [14, 20, 49]. Unlike traditional dataflow systems, in which graph vertices represent functional computation on immutable data, TensorFlow allows vertices to represent computations that own or update mutable state. Edges carry *tensors* (multi-dimensional arrays) between nodes, and TensorFlow transparently inserts the appropriate communication between distributed subcomputations. By unifying the computation and state management in a single programming model, TensorFlow allows programmers to experiment with different parallelization schemes that, for example, offload computation onto the servers that hold the shared state to reduce the amount of network traffic. We have also built various coordination protocols, and achieved encouraging results with synchronous replication, echoing recent results [10, 18] that contradict the commonly held belief that asynchronous replication is required for scalable learning [14, 20, 49].

Over the past year, more than 150 teams at Google have used TensorFlow, and we have released the system as an open-source project.[1] Thanks to our large community of users we have gained experience with many different machine learning applications. In this paper, we focus on neural network training as a challenging systems problem, and select two representative applications from this space: image classification and language modeling. These applications stress computational throughput and aggregate model size respectively, and we use them both to demonstrate the extensibility of TensorFlow, and to evaluate the efficiency and scalability of our present implementation.

2 Background & motivation

We begin by describing the limitations of our previous system (§2.1) and outlining the design principles that we used in the development of TensorFlow (§2.2).

2.1 Previous system: DistBelief

TensorFlow is the successor to DistBelief, which is the distributed system for training neural networks that Google has used since 2011 [20]. DistBelief uses the *parameter server* architecture, and here we criticize its limitations, but other systems based on this architecture have addressed these limitations in other ways [11, 14, 49]; we discuss those systems in Subsection 2.3.

In the parameter server architecture, a job comprises two disjoint sets of processes: stateless *worker* processes that perform the bulk of the computation when training a model, and stateful *parameter server* processes that maintain the current version of the model parameters. DistBelief's programming model is similar to Caffe's [38]: the user defines a neural network as a directed acyclic graph of *layers* that terminates with a *loss function*. A layer is a composition of mathematical operators: for example, a *fully connected* layer multiplies its input by a weight matrix, adds a bias vector, and applies a non-linear function (such as a sigmoid) to the result. A loss function is a scalar function that quantifies the difference between the predicted value (for a given input data point) and the ground truth. In a fully connected layer, the weight matrix and bias vector are *parameters*, which a learning algorithm will update in order to minimize the value of the loss function. DistBelief uses the DAG structure and knowledge of the layers' semantics to compute gradients for each of the model parameters, via backpropagation [63]. Because the parameter updates in many algorithms are commutative and have weak consistency requirements [61], the worker processes can compute updates independently and write back "delta" updates to each parameter server, which combines the updates with its current state.

Although DistBelief has enabled many Google products to use deep neural networks and formed the basis of many machine learning research projects, we soon began to feel its limitations. Its Python-based scripting interface for composing pre-defined layers was adequate for users with simple requirements, but our more advanced users sought three further kinds of flexibility:

Defining new layers For efficiency, we implemented DistBelief layers as C++ classes. Using a separate, less familiar programming language for implementing layers is a barrier for machine learning researchers who seek to experiment with new layer architectures, such as sampled softmax classifiers [37] and attention modules [53].

Refining the training algorithms Many neural networks are trained using stochastic gradient descent (SGD), which iteratively refines the parameters of the network by moving them in the direction that maximally decreases the value of the loss function. Several refinements to SGD accelerate convergence by changing the update rule [23, 66]. Researchers often want to experiment with new optimization methods, but doing that in DistBelief involves modifying the parameter server implementation. Moreover, the `get()` and `put()` interface for the parameter server is not ideal for all optimization methods: sometimes a set of related parameters must be updated atomically, and in many cases it would be more efficient to offload computation onto the parameter server, and thereby reduce the amount of network traffic.

Defining new training algorithms DistBelief workers follow a fixed execution pattern: read a batch of input data and the current parameter values, compute the loss function (a *forward* pass through the network), compute gradients for each of the parameter (a *backward* pass), and write the gradients back to the parameter server. This pattern works for training simple feed-forward neural networks, but fails for more advanced models, such as recurrent neural networks, which contain loops [39]; adversarial networks, in which two related networks are trained alternately [26]; and reinforcement learning models, where the loss function is computed by some agent in a separate system, such as a video game emulator [54]. Moreover, there are many other machine learning algorithms—such as expectation maximization, decision forest training, and latent Dirichlet allocation—that do not fit the same mold as neural network training, but could also benefit from a common, well-optimized distributed runtime.

In addition, we designed DistBelief with a single platform in mind: a large distributed cluster of multicore

[1]Software available from `https://tensorflow.org`.

```
# 1. Construct a graph representing the model.
x = tf.placeholder(tf.float32, [BATCH_SIZE, 784])  # Placeholder for input.
y = tf.placeholder(tf.float32, [BATCH_SIZE, 10])   # Placeholder for labels.

W_1 = tf.Variable(tf.random_uniform([784, 100]))   # 784x100 weight matrix.
b_1 = tf.Variable(tf.zeros([100]))                 # 100-element bias vector.
layer_1 = tf.nn.relu(tf.matmul(x, W_1) + b_2)      # Output of hidden layer.

W_2 = tf.Variable(tf.random_uniform([100, 10]))    # 100x10 weight matrix.
b_2 = tf.Variable(tf.zeros([10]))                  # 10-element bias vector.
layer_2 = tf.matmul(layer_1, W_2) + b_2            # Output of linear layer.

# 2. Add nodes that represent the optimization algorithm.
loss = tf.nn.softmax_cross_entropy_with_logits(layer_2, y)
train_op = tf.train.AdagradOptimizer(0.01).minimize(loss)

# 3. Execute the graph on batches of input data.
with tf.Session() as sess:                         # Connect to the TF runtime.
  sess.run(tf.initialize_all_variables())          # Randomly initialize weights.
  for step in range(NUM_STEPS):                    # Train iteratively for NUM_STEPS.
    x_data, y_data = ...                           # Load one batch of input data.
    sess.run(train_op, {x: x_data, y: y_data})     # Perform one training step.
```

Figure 1: An image classifier written using TensorFlow's Python API. This program is a simple solution to the MNIST digit classification problem [48], with 784-pixel images and 10 output classes.

servers [20]. We were able to add support for GPU acceleration, when it became clear that this acceleration would be crucial for executing convolutional kernels efficiently [44], but DistBelief remains a heavyweight system that is geared for training deep neural networks on huge datasets, and is difficult to scale *down* to other environments. In particular, many users want to hone their model locally on a GPU-powered workstation, before scaling the same code to train on a much larger dataset. After training a model on a cluster, the next step is to push the model into production, which might involve integrating the model into an online service, or deploying it onto a mobile device for offline execution. Each of these tasks has some common computational structure, but our colleagues found it necessary to use or create separate systems that satisfy the different performance and resource requirements of each platform. TensorFlow provides a single programming model and runtime system for all of these environments.

2.2 Design principles

We designed TensorFlow to be much more flexible than DistBelief, while retaining its ability to satisfy the demands of Google's production machine learning workloads. TensorFlow provides a simple dataflow-based programming abstraction that allows users to deploy applications on distributed clusters, local workstations, mobile devices, and custom-designed accelerators. A high-level scripting interface (Figure 1) wraps the construction of dataflow graphs and enables users to experiment with different model architectures and optimization algorithms without modifying the core system. In this subsection, we briefly highlight TensorFlow's core design principles:

Dataflow graphs of primitive operators Both TensorFlow and DistBelief use a dataflow representation for their models, but the most striking difference is that a DistBelief model comprises relatively few complex "layers", whereas the corresponding TensorFlow model represents individual mathematical operators (such as matrix multiplication, convolution, etc.) as nodes in the dataflow graph. This approach makes it easier for users to compose novel layers using a high-level scripting interface. Many optimization algorithms require each layer to have defined gradients, and building layers out of simple operators makes it easy to differentiate these models automatically (§4.1). In addition to the functional operators, we represent mutable state, and the operations that update it, as nodes in the dataflow graph, thus enabling experimentation with different update rules.

Deferred execution A typical TensorFlow application has two distinct phases: the first phase defines the program (e.g., a neural network to be trained and the update rules) as a symbolic dataflow graph with placeholders for the input data and variables that represent the state; and the second phase executes an optimized version of the program on the set of available devices. By deferring the execution until the entire program is available, TensorFlow can optimize the execution phase by using global information about the computation. For example, TensorFlow achieves high GPU utilization by using the graph's dependency structure to issue a sequence of kernels to the GPU without waiting for intermediate results. While this design choice makes execution more efficient, we have had to push more complex features—such as dynamic control flow (§3.4)—into the dataflow graph, so that models using these features enjoy the same optimizations.

Common abstraction for heterogeneous accelerators In addition to general-purpose devices such as multicore CPUs and GPUs, special-purpose accelerators for deep learning can achieve significant performance improvements and power savings. At Google, our colleagues have built the Tensor Processing Unit (TPU) specifically for machine learning; TPUs yield an order of magnitude improvement in performance-per-watt compared to alternative state-of-the-art technology [40]. To support these accelerators in TensorFlow, we define a common abstraction for devices. At a minimum, a device must implement methods for (i) issuing a kernel for execution, (ii) allocating memory for inputs and outputs, and (iii) transferring buffers to and from host memory. Each operator (e.g., matrix multiplication) can have multiple specialized implementations for different devices. As a result, the same program can easily target GPUs, TPUs, or mobile CPUs as required for training, serving, and offline inference.

TensorFlow uses tensors of primitive values as a common interchange format that all devices understand. At the lowest level, all tensors in TensorFlow are dense; sparse tensors can be represented in terms of dense ones (§3.1). This decision ensures that the lowest levels of the system have simple implementations for memory allocation and serialization, thus reducing the framework overhead. Tensors also enable other optimizations for memory management and communication, such as RDMA and direct GPU-to-GPU transfer.

The main consequence of these principles is that in TensorFlow there is no such thing as a parameter server. On a cluster, we deploy TensorFlow as a set of *tasks* (named processes that can communicate over a network) that each export the same graph execution API and contain one or more devices. Typically a subset of those tasks assumes the role that a parameter server plays in other systems [11, 14, 20, 49], and we therefore call them *PS tasks*; the others are *worker tasks*. However, since a PS task is capable of running arbitrary TensorFlow graphs, it is more flexible than a conventional parameter server: users can program it with the same scripting interface that they use to define models. This flexibility is the key difference between TensorFlow and contemporary systems, and in the rest of the paper we will discuss some of the applications that this flexibility enables.

2.3 Related work

Single-machine frameworks Many machine learning researchers carry out their work on a single—often GPU-equipped—computer [43, 44], and several single-machine frameworks support this scenario. Caffe [38] is a high-performance framework for training declaratively specified neural networks on multicore CPUs and GPUs. As discussed above, its programming model is similar to DistBelief (§2.1), so it is easy to compose models from existing layers, but relatively difficult to add new layers or optimizers. Theano [2] allows programmers to express a model as a dataflow graph of primitive operators, and generates efficient compiled code for training that model. Its programming model is closest to TensorFlow, and it provides much of the same flexibility in a single machine.

Unlike Caffe, Theano, and TensorFlow, Torch [17] offers a powerful *imperative* programming model for scientific computation and machine learning. It allows fine-grained control over the execution order and memory utilization, which enables power users to optimize the performance of their programs. While this flexibility is useful for research, Torch lacks the advantages of a dataflow graph as a portable representation across small-scale experimentation, production training, and deployment.

Batch dataflow systems Starting with MapReduce [21], batch dataflow systems have been applied to a large number of machine learning algorithms [70], and more recent systems have focused on increasing expressivity and performance. DryadLINQ [74] adds a high-level query language that supports more sophisticated algorithms than MapReduce. Spark [75] extends DryadLINQ with the ability to cache previously computed datasets in memory, and is therefore better suited to iterative machine learning algorithms (such as k-means clustering and logistic regression) when the input data fit in memory. Dandelion extends DryadLINQ with code generation for GPUs [62] and FPGAs [16].

The principal limitation of a batch dataflow system is that it requires the input data to be immutable, and all of the subcomputations to be deterministic, so that the system can re-execute subcomputations when machines in the cluster fail. This feature—which is beneficial for many conventional workloads—makes updating a ma-

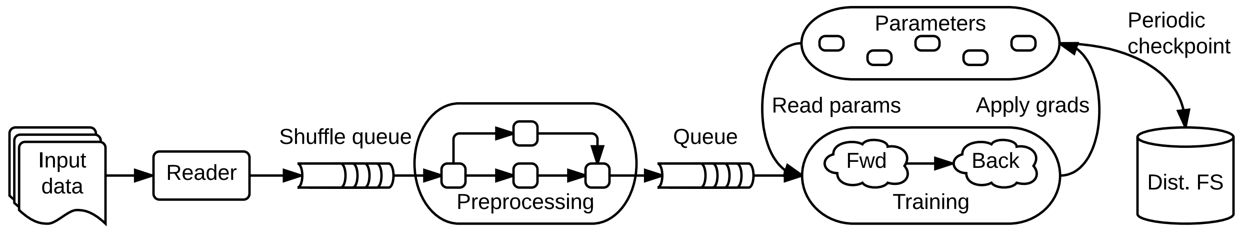


Figure 2: A schematic TensorFlow dataflow graph for a training pipeline, containing subgraphs for reading input data, preprocessing, training, and checkpointing state.

chine learning model an expensive operation. For example, the SparkNet system for training deep neural networks on Spark takes 20 seconds to broadcast weights and collect updates from five workers [55]. As a result, in these systems, each model update step must process larger batches, slowing convergence [8]. We show in Subsection 6.3 that TensorFlow can train larger models on larger clusters with step times as short as 2 seconds.

Parameter servers As we discuss in Subsection 2.1, a parameter server architecture uses a set of servers to manage shared state that is updated by a set of parallel workers. This architecture emerged in work on scalable topic modeling [65], and DistBelief showed how it can apply to deep neural network training. Project Adam [14] further applied this architecture for the efficient training of convolutional neural networks; and Li *et al.*'s "Parameter Server" [49] added innovations in consistency models, fault tolerance, and elastic rescaling. Despite earlier skepticism that parameter servers would be compatible with GPU acceleration [14], Cui *et al.* recently showed that a parameter server specialized for use with GPUs can achieve speedups on small clusters [18].

MXNet [11] is perhaps the closest system in design to TensorFlow. It uses a dataflow graph to represent the computation at each worker, and uses a parameter server to scale training across multiple machines. The MXNet parameter server exports a key-value store interface that supports aggregating updates sent from multiple devices in each worker, and using an arbitrary user-provided function to combine incoming updates with the current value. The MXNet key-value store interface [22] does not currently allow sparse gradient updates within a single value, which are crucial for the distributed training of large models (§4.2), and adding this feature would require modifications to the core system.

The parameter server architecture meets many of our requirements, and with sufficient engineering effort it would be possible to build most of the features that we describe in this paper into a parameter server. For TensorFlow we sought a high-level programming model that allows users to customize the code that runs in all parts of the system, so that the cost of experimentation with new optimization algorithms and model architectures is lower. In the next section, we describe the building blocks of a TensorFlow program in more detail.

3 TensorFlow execution model

TensorFlow uses a single dataflow graph to represent all computation and state in a machine learning algorithm, including the individual mathematical operations, the parameters and their update rules, and the input preprocessing (Figure 2). The dataflow graph expresses the communication between subcomputations explicitly, thus making it easy to execute independent computations in parallel and to partition computations across multiple devices. TensorFlow differs from batch dataflow systems (§2.3) in two respects:

- The model supports multiple concurrent executions on overlapping subgraphs of the overall graph.
- Individual vertices may have mutable state that can be shared between different executions of the graph.

The key observation in the parameter server architecture [14, 20, 49] is that mutable state is crucial when training very large models, because it becomes possible to make in-place updates to very large parameters, and propagate those updates to parallel training steps as quickly as possible. Dataflow with mutable state enables TensorFlow to mimic the functionality of a parameter server, but with additional flexibility, because it becomes possible to execute arbitrary dataflow subgraphs on the machines that host the shared model parameters. As a result, our users have been able to experiment with different optimization algorithms, consistency schemes, and parallelization strategies.

3.1 Dataflow graph elements

In a TensorFlow graph, each vertex represents a unit of local computation, and each edge represents the output from, or input to, a vertex. We refer to the computation at vertices as *operations*, and the values that flow along edges as *tensors*. In this subsection, we describe the common types of operations and tensors.

Tensors In TensorFlow, we model all data as tensors (n-dimensional arrays) with the elements having one of a small number of primitive types, such as `int32`, `float32`, or `string` (where `string` can represent arbitrary binary data). Tensors naturally represent the inputs to and results of the common mathematical operations in many machine learning algorithms: for example, a matrix multiplication takes two 2-D tensors and produces a 2-D tensor; and a batch 2-D convolution takes two 4-D tensors and produces another 4-D tensor.

At the lowest level, all TensorFlow tensors are dense, for the reasons we discuss in Subsection 2.2. TensorFlow offers two alternatives for representing sparse data: either encode the data into variable-length `string` elements of a dense tensor, or use a tuple of dense tensors (e.g., an n-D sparse tensor with m non-zero elements can be represented in coordinate-list format as an $m \times n$ matrix of coordinates and a length-m vector of values). The shape of a tensor can vary in one or more of its dimensions, which makes it possible to represent sparse tensors with differing numbers of elements.

Operations An operation takes $m \geq 0$ tensors as input and produces $n \geq 0$ tensors as output. An operation has a named "type" (such as `Const`, `MatMul`, or `Assign`) and may have zero or more compile-time attributes that determine its behavior. An operation can be polymorphic and variadic at compile-time: its attributes determine both the expected types and arity of its inputs and outputs.

For example, the simplest operation `Const` has no inputs and a single output; its value is a compile-time attribute. For example, `AddN` sums multiple tensors of the same element type, and it has a type attribute `T` and an integer attribute `N` that define its type signature.

Stateful operations: variables An operation can contain mutable state that is read and/or written each time it executes. A `Variable` operation owns a mutable buffer that may be used to store the shared parameters of a model as it is trained. A `Variable` has no inputs, and produces a *reference handle*, which acts as a typed capability for reading and writing the buffer. A `Read` operation takes a reference handle r as input, and outputs the value of the variable ($\mathrm{State}[r]$) as a dense tensor. Other operations modify the underlying buffer: for example, `AssignAdd` takes a reference handle r and a tensor value x, and when executed performs the update $\mathrm{State}'[r] \leftarrow \mathrm{State}[r] + x$. Subsequent `Read`$(r)$ operations produce the value $\mathrm{State}'[r]$.

Stateful operations: queues TensorFlow includes several queue implementations, which support more advanced forms of coordination. The simplest queue is `FIFOQueue`, which owns an internal queue of tensors, and allows concurrent access in first-in-first-out order. Other types of queues dequeue tensors in random and priority orders, which ensure that input data are sampled appropriately. Like a `Variable`, the `FIFOQueue` operation produces a reference handle that can be consumed by one of the standard queue operations, such as `Enqueue` and `Dequeue`. These operations push their input onto the tail of the queue and, respectively, pop the head element and output it. `Enqueue` will block if its given queue is full, and `Dequeue` will block if its given queue is empty. When queues are used in an input preprocessing pipeline, this blocking provides backpressure; it also supports synchronization (§4.4). The combination of queues and dynamic control flow (§3.4) can also implement a form of streaming computation between subgraphs.

3.2 Partial and concurrent execution

TensorFlow uses a dataflow graph to represent all possible computations in a particular application. The API for executing a graph allows the client to specify declaratively the *subgraph* that should be executed. The client selects zero or more edges to *feed* input tensors into the dataflow, and one or more edges to *fetch* output tensors from the dataflow; the runtime then prunes the graph to contain the necessary set of operations. Each invocation of the API is called a *step*, and TensorFlow supports multiple *concurrent steps* on the same graph. Stateful operations allow steps to share data and synchronize when necessary.

Figure 2 shows a typical training application, with multiple subgraphs that execute concurrently and interact through shared variables and queues. The core training subgraph depends on a set of model parameters and on input batches from a queue. Many concurrent steps of the training subgraph update the model based on different input batches, to implement data-parallel training. To fill the input queue, concurrent preprocessing steps transform individual input records (e.g., decoding images and applying random distortions), and a separate I/O subgraph reads records from a distributed file system. A checkpointing subgraph runs periodically for fault tolerance (§4.3).

Partial and concurrent execution is responsible for much of TensorFlow's flexibility. Adding mutable state and coordination via queues makes it possible to specify a wide variety of model architectures in user-level code, which enables advanced users to experiment without modifying the internals of the TensorFlow runtime. By default, concurrent executions of a TensorFlow subgraph run asynchronously with respect to one another. This asynchrony makes it straightforward to implement machine learning algorithms with weak consistency requirements [61], which include many neural network training algorithms [20]. As we discuss later, TensorFlow also provides the primitives needed to synchronize workers during training (§4.4), which has led to promising results on some learning tasks (§6.3).

3.3 Distributed execution

Dataflow simplifies distributed execution, because it makes communication between subcomputations explicit. It enables the same TensorFlow program to be deployed to a cluster of GPUs for training, a cluster of TPUs for serving, and a cellphone for mobile inference.

Each operation resides on a particular *device*, such as a CPU or GPU in a particular *task*. A device is responsible for executing a *kernel* for each operation assigned to it. TensorFlow allows multiple kernels to be registered for a single operation, with specialized implementations for a particular device or data type (see §5 for details). For many operations, such as element-wise operators (`Add`, `Sub`, etc.), we can compile a single kernel implementation for CPU and GPU using different compilers.

The TensorFlow runtime places operations on devices, subject to implicit or explicit constraints in the graph. The placement algorithm computes a feasible set of devices for each operation, calculates the sets of operations that must be colocated, and selects a satisfying device for each colocation group. It respects implicit colocation constraints that arise because each stateful operation and its state must be placed on the same device. In addition, the user may specify partial device preferences such as "any device in a particular task", or "a GPU in any task", and the runtime will respect these constraints. A typical training application will use client-side programming constructs to add constraints such that, for example, parameters are distributed among a set of "PS" tasks (§4.2).

TensorFlow thus permits great flexibility in how operations in the dataflow graph are mapped to devices. While simple heuristics yield adequate performance for novice users, expert users can optimize performance by manually placing operations to balance the computation, memory, and network requirements across multiple tasks and multiple devices within those tasks. An open question is how

```
input = ...  # A sequence of tensors
state = 0    # Initial state
w = ...      # Trainable weights

for i in range(len(input)):
  state, out[i] = f(state, w, input[i])
```

Figure 3: Pseudocode for an abstract RNN (§3.4). The function `f` typically comprises differentiable operations such as matrix multiplications and convolutions [32]. TensorFlow implements the loop in its dataflow graph.

TensorFlow can automatically determine placements that achieve close to optimal performance on a given set of devices, thus freeing users from this concern. Even without such automation, it may be worthwhile to separate placement directives from other aspects of model definitions, so that, for example, it would be trivial to modify placements after a model has been trained.

Once the operations in a graph have been placed, and the partial subgraph has been computed for a step (§3.2), TensorFlow partitions the operations into per-device subgraphs. A per-device subgraph for device d contains all of the operations that were assigned to d, with additional `Send` and `Recv` operations that replace edges across device boundaries. `Send` transmits its single input to a specified device as soon as the tensor is available, using a *rendezvous key* to name the value. `Recv` has a single output, and blocks until the value for a specified rendezvous key is available locally, before producing that value. `Send` and `Recv` have specialized implementations for several device-type pairs; we describe some of these in Section 5.

We optimized TensorFlow for executing large subgraphs repeatedly with low latency. Once the graph for a step has been pruned, placed, and partitioned, its subgraphs are cached in their respective devices. A client *session* maintains the mapping from step definitions to cached subgraphs, so that a distributed step on a large graph can be initiated with one small message to each participating task. This model favors static, reusable graphs, but it can support dynamic computations using dynamic control flow, as the next subsection describes.

3.4 Dynamic control flow

TensorFlow supports advanced machine learning algorithms that contain conditional and iterative control flow. For example, a *recurrent neural network* (RNN) [39] such as an LSTM [32] can generate predictions from sequential data. Google's Neural Machine Translation system uses TensorFlow to train a deep LSTM that achieves state-of-the-art performance on many translation tasks [73]. The core of an RNN is a recurrence relation, where the output for sequence element i is a function of some state that accumulates across the sequence (Figure 3). In this case, dynamic control flow enables iteration over sequences that have variable lengths, without unrolling the computation to the length of the longest sequence.

As we discussed in Subsection 2.2, TensorFlow uses deferred execution via the dataflow graph to offload larger chunks of work to accelerators. Therefore, to implement RNNs and other advanced algorithms, we add conditional (if statement) and iterative (while loop) programming constructs in the dataflow graph itself. We use these primitives to build higher-order constructs, such as `map()`, `fold()`, and `scan()` [2].

For this purpose, we borrow the `Switch` and `Merge` primitives from classic dynamic dataflow architectures [4]. `Switch` is a demultiplexer: it takes a data input and a control input, and uses the control input to select which of its two outputs should produce a value. The `Switch` output not taken receives a special *dead* value, which propagates recursively through the rest of the graph until it reaches a `Merge` operation. `Merge` is a multiplexer: it forwards at most one non-dead input to its output, or produces a dead output if both of its inputs are dead. The conditional operator uses `Switch` to execute one of two branches based on the runtime value of a boolean tensor, and `Merge` to combine the outputs of the branches. The while loop is more complicated, and uses `Enter`, `Exit`, and `NextIteration` operators to ensure that the loop is well-formed [56].

The execution of iterations can overlap, and TensorFlow can also partition conditional branches and loop bodies across multiple devices and processes. The partitioning step adds logic to coordinate the start and termination of each iteration on each device, and to decide the termination of the loop. As we will see in Subsection 4.1, TensorFlow also supports automatic differentiation of control flow constructs. Automatic differentiation adds the subgraphs for computing gradients to the dataflow graph, which TensorFlow partitions across potentially distributed devices to compute the gradients in parallel.

4 Extensibility case studies

By choosing a unified representation for all computation in TensorFlow, we enable users to experiment with features that were hard-coded into the DistBelief runtime. In this section, we discuss four extensions that we have built using dataflow primitives and "user-level" code.

4.1 Differentiation and optimization

Many learning algorithms train a set of parameters using some variant of SGD, which entails computing the *gradients* of a loss function with respect to those parameters, then updating the parameters based on those gradients. TensorFlow includes a user-level library that differentiates a symbolic expression for a loss function and produces a new symbolic expression representing the gradients. For example, given a neural network as a composition of layers and a loss function, the library will automatically derive the backpropagation code.

The differentiation algorithm performs breadth-first search to identify all of the backwards paths from the target operation (e.g., a loss function) to a set of parameters, and sums the partial gradients that each path contributes. Our users frequently specialize the gradients for some operations, and they have implemented optimizations like batch normalization [33] and gradient clipping [60] to accelerate training and make it more robust. We have extended the algorithm to differentiate conditional and iterative subcomputations (§3.4) by adding nodes to the graph that record the control flow decisions in the forward pass, and replaying those decisions in reverse during the backward pass. Differentiating iterative computations over long sequences can lead to a large amount of intermediate state being accumulated in memory, and we have developed techniques for managing limited GPU memory on these computations.

TensorFlow users can also experiment with a wide range of *optimization algorithms*, which compute new values for the parameters in each training step. SGD is easy to implement in a parameter server: for each parameter W, gradient $\partial L/\partial W$, and learning rate α, the update rule is $W' \leftarrow W - \alpha \times \partial L/\partial W$. A parameter server can implement SGD by using $-=$ as the write operation, and writing $\alpha \times \partial L/\partial W$ to each W after a training step.

However, there are many more advanced optimization schemes that are difficult to express as a single write operation. For example, the Momentum algorithm accumulates a "velocity" for each parameter based on its gradient over multiple iterations, then computes the parameter update from that accumulation; and many refinements to this algorithm have been proposed [66]. Implementing Momentum in DistBelief [20], required modifications to the parameter server implementation to change the representation of parameter data, and execute complex logic in the write operation; such modifications are challenging for many users. Optimization algorithms are the topic of active research, and researchers have implemented several on top of TensorFlow, including Momentum, AdaGrad, AdaDelta, RMSProp, Adam, and L-BFGS. These

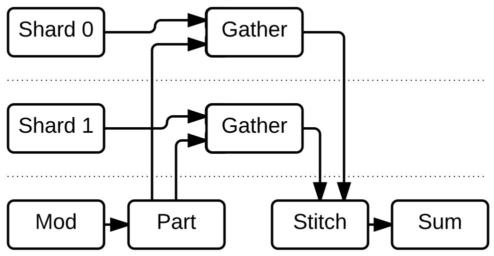


Figure 4: Schematic dataflow for an embedding layer (§4.2) with a two-way sharded embedding matrix.

can be built in TensorFlow using `Variable` operations and primitive mathematical operations without modifying the underlying system, so it is easy to experiment with new algorithms as they emerge.

4.2 Training very large models

To train a model on high-dimensional data, such as words in a corpus of text [7], it is common to use a *distributed representation*, which embeds a training example as a pattern of activity across several neurons, and which can be learned by backpropagation [30]. For example, in a language model, a training example might be a sparse vector with non-zero entries corresponding to the IDs of words in a vocabulary, and the distributed representation for each word will be a lower-dimensional vector [6]. "Wide and deep learning" creates distributed representations from cross-product transformations on categorical features, and the implementation on TensorFlow is used to power the Google Play app store recommender system [12].

Inference begins by multiplying a batch of b sparse vectors against an $n \times d$ *embedding matrix*, where n is the number of words in the vocabulary, and d is the desired dimensionality, to produce a much smaller $b \times d$ dense matrix representation; for training, most optimization algorithms modify only the rows of the embedding matrix that were read by the sparse multiplication. In TensorFlow models that process sparse data, $n \times d$ can amount to gigabytes of parameters: e.g., a large language model may use over 10^9 parameters with a vocabulary of 800,000 words [41], and we have experience with document models [19] where the parameters occupy several terabytes. Such models are too large to copy to a worker on every use, or even to store in RAM on a single host.

We implement sparse embedding layers in the TensorFlow graph as a composition of primitive operations. Figure 4 shows a simplified graph for an embedding layer that is split across two parameter server tasks. The core operation of this subgraph is `Gather`, which extracts a sparse set of rows from a tensor, and TensorFlow colocates this operation with the variable on which it operates. The dynamic partition (`Part`) operation divides the incoming indices into variable-sized tensors that contain the indices destined for each shard, and the dynamic stitching (`Stitch`) operation reassembles the partial results from each shard into a single result tensor. Each of these operations has a corresponding gradient, so it supports automatic differentiation (§4.1), and the result is a set of sparse update operations that act on just the values that were originally gathered from each of the shards.

Users writing a TensorFlow model typically do not construct graphs like Figure 4 manually. Instead TensorFlow includes libraries that expose the abstraction of a sharded parameter, and build appropriate graphs of primitive operations based on the desired degree of distribution.

While sparse reads and updates are possible in a parameter server [49], TensorFlow adds the flexibility to offload arbitrary computation onto the devices that host the shared parameters. For example, classification models typically use a softmax classifier that multiplies the final output by a weight matrix with c columns, where c is the number of possible classes; for a language model, c is the size of the vocabulary, which can be large. Our users have experimented with several schemes to accelerate the softmax calculation. The first is similar to an optimization in Project Adam [14], whereby the weights are sharded across several tasks, and the multiplication and gradient calculation are colocated with the shards. More efficient training is possible using a *sampled softmax* [37], which performs a sparse multiplication based on the true class for an example and a set of randomly sampled false classes. We compare the performance of these two schemes in §6.4.

4.3 Fault tolerance

Training a model can take several hours or days, even using a large number of machines [14, 20]. We often need to train a model using non-dedicated resources, for example using the Borg cluster manager [71], which does not guarantee availability of the same resources for the duration of the training process. Therefore, a long-running TensorFlow job is likely to experience failure or pre-emption, and we require some form of fault tolerance. It is unlikely that tasks will fail so often that individual operations need fault tolerance, so a mechanism like Spark's RDDs [75] would impose significant overhead for little benefit. There is no need to make every write to the parameter state durable, because we can recompute any update from the input data, and many learning algorithms do not require strong consistency [61].

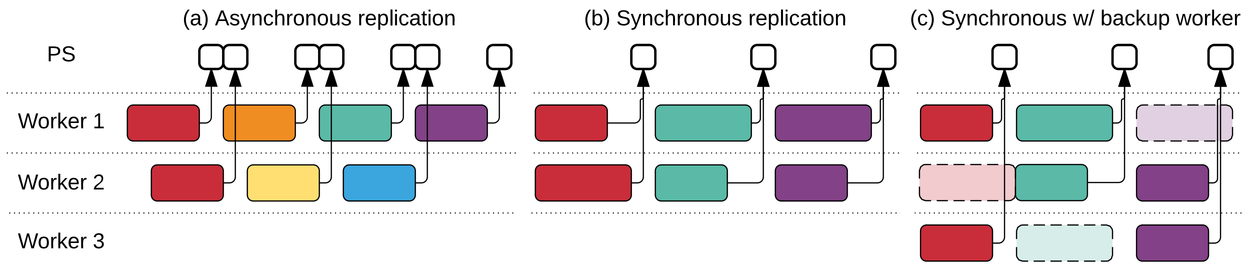

(a) Asynchronous replication
(b) Synchronous replication
(c) Synchronous w/ backup worker
PS
Worker 1
Worker 2
Worker 3

Figure 5: Three synchronization schemes for parallel SGD. Each color represents a different starting parameter value; a white square is a parameter update. In (c), a dashed rectangle represents a backup worker whose result is discarded.

We implement user-level checkpointing for fault tolerance, using two operations in the graph (Figure 2): `Save` writes one or more tensors to a checkpoint file, and `Restore` reads one or more tensors from a checkpoint file. Our typical configuration connects each `Variable` in a task to the same `Save` operation, with one `Save` per task, to maximize the I/O bandwidth to a distributed file system. The `Restore` operations read named tensors from a file, and a standard `Assign` stores the restored value in its respective variable. During training, a typical client runs all of the `Save` operations periodically to produce a new checkpoint; when the client starts up, it attempts to `Restore` the latest checkpoint.

TensorFlow includes a client library for constructing the appropriate graph structure and for invoking `Save` and `Restore` as necessary. This behavior is customizable: the user can apply different policies to subsets of the variables in a model, or customize the checkpoint retention scheme. For example, many users retain checkpoints with the highest score in a custom evaluation metric. The implementation is also reusable: it may be used for model fine-tuning and unsupervised pre-training [45, 47], which are forms of transfer learning, in which the parameters of a model trained on one task (e.g., recognizing general images) are used as the starting point for another task (e.g., recognizing breeds of dog). Having checkpoint and parameter management as programmable operations in the graph gives users the flexibility to implement schemes like these and others that we have not anticipated.

The checkpointing library does not attempt to produce consistent checkpoints: if training and checkpointing execute concurrently, the checkpoint may include none, all, or some of the updates from the training step. This behavior is compatible with the relaxed guarantees of asynchronous SGD [20]. Consistent checkpoints require additional synchronization to ensure that update operations do not interfere with checkpointing; if desired, one can use the scheme in the next subsection to take a checkpoint after the synchronous update step.

4.4 Synchronous replica coordination

SGD is robust to asynchrony [61], and many systems train deep neural networks using asynchronous parameter updates [14, 20], which are believed scalable because they maintain high throughput in the presence of stragglers. The increased throughput comes at the cost of using stale parameter values in training steps. Some have recently revisited the assumption that *synchronous* training does not scale [10, 18]. Since GPUs enable training with hundreds—rather than thousands [47]—of machines, synchronous training may be faster (in terms of time to quality) than asynchronous training on the same platform.

Though we originally designed TensorFlow for asynchronous training, we have begun experimenting with synchronous methods. The TensorFlow graph enables users to change how parameters are read and written when training a model, and we implement three alternatives. In the asynchronous case (Figure 5(a)), each worker reads the current values of parameters when each step begins, and applies its gradient to the (possibly different) current values at the end: this approach ensures high utilization, but the individual steps use stale parameter values, making each step less effective. We implement the synchronous version using queues (§3.1) to coordinate execution: a blocking queue acts as a barrier to ensure that all workers read the same parameter values, and a per-variable queue accumulates gradient updates from all workers in order to apply them atomically. The simple synchronous version (Figure 5(b)) accumulates updates from all workers before applying them, but slow workers limit overall throughput.

To mitigate stragglers, we implement *backup workers* (Figure 5(c), [10]), which are similar to MapReduce backup tasks [21]. Whereas MapReduce starts backup tasks reactively—after detecting a straggler—our backup workers run proactively, and the aggregation takes the first m of n updates produced. We exploit the fact that SGD samples training data randomly at each step, so each worker processes a different random batch, and it is not a

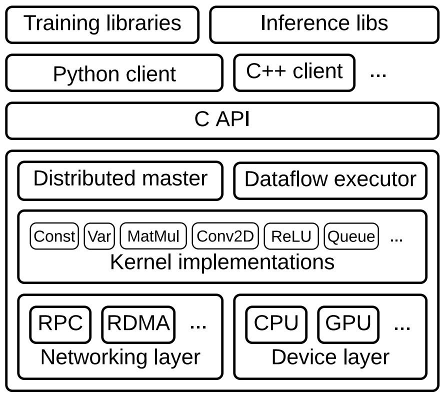


Figure 6: The layered TensorFlow architecture.

problem if a particular batch is ignored. In §6.3 we show how backup workers improve throughput by up to 10%.

5 Implementation

The TensorFlow runtime is a cross-platform library. Figure 6 illustrates its architecture: a C API separates user-level code in different languages from the core runtime.

The core TensorFlow library is implemented in C++ for portability and performance: it runs on several operating systems including Linux, Mac OS X, Windows, Android, and iOS; the x86 and various ARM-based CPU architectures; and NVIDIA's Kepler, Maxwell, and Pascal GPU microarchitectures. The implementation is open-source, and we have accepted several external contributions that enable TensorFlow to run on other architectures.

The *distributed master* translates user requests into execution across a set of tasks. Given a graph and a step definition, it prunes (§3.2) and partitions (§3.3) the graph to obtain subgraphs for each participating device, and caches these subgraphs so that they may be re-used in subsequent steps. Since the master sees the overall computation for a step, it applies standard optimizations such as common subexpression elimination and constant folding; pruning is a form of dead code elimination. It then coordinates execution of the optimized subgraphs across a set of tasks.

The *dataflow executor* in each task handles requests from the master, and schedules the execution of the kernels that comprise a local subgraph. We optimize the dataflow executor for running large graphs with low overhead. Our current implementation can execute 10,000 subgraphs per second (§6.2), which enables a large number of replicas to make rapid, fine-grained training steps. The dataflow executor dispatches kernels to local devices and runs kernels in parallel when possible, for example by using multiple CPU cores or GPU streams.

The runtime contains over 200 standard operations, including mathematical, array manipulation, control flow, and state management operations. Many of the operation kernels are implemented using Eigen::Tensor [36], which uses C++ templates to generate efficient parallel code for multicore CPUs and GPUs; however, we liberally use libraries like cuDNN [13] where a more efficient kernel implementation is possible. We have also implemented *quantization*, which enables faster inference in environments such as mobile devices and high-throughput datacenter applications, and use the `gemmlowp` low-precision matrix library [35] to accelerate quantized computation.

We specialize `Send` and `Recv` operations for each pair of source and destination device types. Transfers between local CPU and GPU devices use the `cudaMemcpyAsync()` API to overlap computation and data transfer; transfers between two local GPUs use DMA to relieve pressure on the host. For transfers between tasks, TensorFlow uses multiple protocols, including gRPC over TCP, and RDMA over Converged Ethernet. We are also investigating optimizations for GPU-to-GPU communication that use collective operations [59].

Section 4 describes features that we implement completely above the C API, in user-level code. Typically, users compose standard operations to build higher-level abstractions, such as neural network layers, optimization algorithms (§4.1), and sharded embedding computations (§4.2). TensorFlow supports multiple client languages, and we have prioritized Python and C++, because our internal users are most familiar with these languages. As features become more established, we typically port them to C++, so that users can access an optimized implementation from all client languages.

If it is difficult or inefficient to represent a subcomputation as a composition of operations, users can register additional kernels that provide an efficient implementation written in C++. We have found it profitable to hand-implement *fused kernels* for some performance critical operations, such as the ReLU and Sigmoid activation functions and their corresponding gradients. We are currently investigating automatic kernel fusion using a compilation-based approach.

In addition to the core runtime, our colleagues have built several tools that aid users of TensorFlow. These include serving infrastructure for inference in production [27], a visualization dashboard that enables users to follow the progress of a training run, a graph visualizer that helps users to understand the connections in a model, and a distributed profiler that traces the execution of a computation across multiple devices and tasks. We describe these tools in an extended whitepaper [1].

6 Evaluation

In this section, we evaluate the performance of TensorFlow on several synthetic and realistic workloads. Unless otherwise stated, we run all experiments on a shared production cluster, and all figures plot median values with error bars showing the 10th and 90th percentiles.

In this paper we focus on system performance metrics, rather than learning objectives like time to accuracy. TensorFlow is a system that allows machine learning practitioners and researchers to experiment with new techniques, and this evaluation demonstrates that the system (i) has little overhead, and (ii) can employ large amounts of computation to accelerate real-world applications. While techniques like synchronous replication can enable some models to converge in fewer steps overall, we defer the analysis of such improvements to other papers.

6.1 Single-machine benchmarks

Although TensorFlow is a system for "large-scale" machine learning, it is imperative that scalability does not mask poor performance at small scales [51]. Table 1 contains results from Chintala's benchmark of convolutional models on TensorFlow and three single-machine frameworks [15]. All frameworks use a six-core Intel Core i7-5930K CPU at 3.5 GHz and an NVIDIA Titan X GPU.

Library	Training step time (ms)			
	AlexNet	Overfeat	OxfordNet	GoogleNet
Caffe [38]	324	823	1068	1935
Neon [58]	87	**211**	**320**	**270**
Torch [17]	**81**	268	529	470
TensorFlow	**81**	279	540	445

Table 1: Step times for training four convolutional models with different libraries, using one GPU. All results are for training with 32-bit floats. The fastest time for each model is shown in bold.

Table 1 shows that TensorFlow achieves shorter step times than Caffe [38], and performance within 6% of the latest version of Torch [17]. We attribute the similar performance of TensorFlow and Torch to the fact that both use the same version of the cuDNN library [13], which implements the convolution and pooling operations on the critical path for training; Caffe uses open-source implementations for these operations that are simpler but less efficient than cuDNN. The Neon library [58] outperforms TensorFlow on three of the models, by using hand-optimized convolutional kernels [46] implemented in assembly language; in principle, we could follow the same approach in TensorFlow, but we have not yet done so.

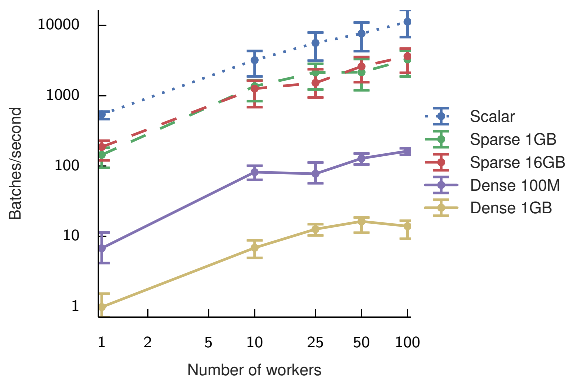


Figure 7: Baseline throughput for synchronous replication with a null model. Sparse accesses enable TensorFlow to handle larger models, such as embedding matrices (§4.2).

6.2 Synchronous replica microbenchmark

The performance of our coordination implementation (§4.4) is the main limiting factor for scaling with additional machines. Figure 7 shows that number of *null training steps* that TensorFlow performs per second for varying model sizes, and increasing numbers of *synchronous* workers. In a null training step, a worker fetches the shared model parameters from 16 PS tasks, performs a trivial computation, and sends updates to the parameters.

The *Scalar* curve in Figure 7 shows the best performance that we could expect for a synchronous training step, because only a single 4-byte value is fetched from each PS task. The median step time is 1.8 ms using a single worker, growing to 8.8 ms with 100 workers. These times measure the overhead of the synchronization mechanism, and capture some of the noise that we expect when running on a shared cluster.

The *Dense* curves show the performance of a null step when the worker fetches the entire model. We repeat the experiment with models of size 100 MB and 1 GB, with the parameters sharded equally over 16 PS tasks. The median step time for 100 MB increases from 147 ms with one worker to 613 ms with 100 workers. For 1 GB, it increases from 1.01 s with one worker to 7.16 s with 100 workers.

For large models, a typical training step accesses only a subset of the parameters, and the *Sparse* curves show the throughput of the embedding lookup operation from Subsection 4.2. Each worker reads 32 randomly selected entries from a large embedding matrix containing 1 GB or 16 GB of data. As expected, the step times do not vary with the size of the embedding, and TensorFlow achieves step times ranging from 5 to 20 ms.

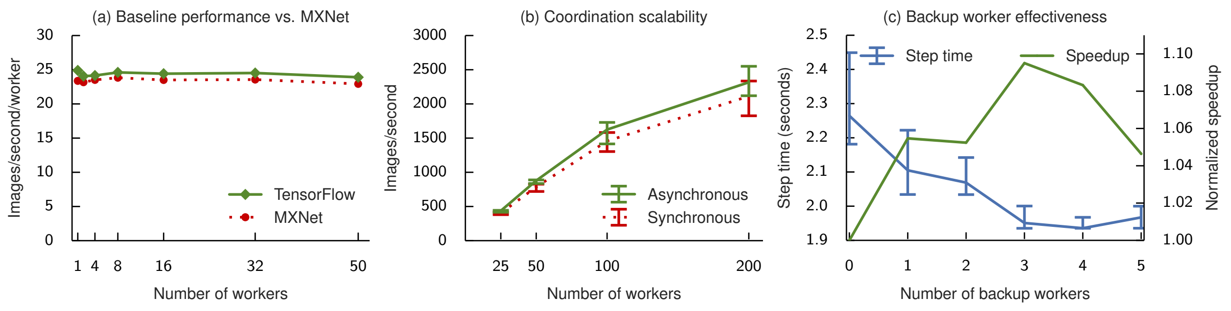


Figure 8: Results of the performance evaluation for Inception-v3 training (§6.3). (a) TensorFlow achieves slightly better throughput than MXNet for asynchronous training. (b) Asynchronous and synchronous training throughput increases with up to 200 workers. (c) Adding backup workers to a 50-worker training job can reduce the overall step time, and improve performance even when normalized for resource consumption.

6.3 Image classification

Deep neural networks have achieved breakthrough performance on computer vision tasks such as recognizing objects in photographs [44], and these tasks are a key application for TensorFlow at Google. Training a network to high accuracy requires a large amount of computation, and we use TensorFlow to scale out this computation across a cluster of GPU-enabled servers. In these experiments, we focus on Google's Inception-v3 model, which achieves 78.8% accuracy in the ILSVRC 2012 image classification challenge [69]; the same techniques apply to other deep convolutional models—such as ResNet [28]—implemented on TensorFlow. We investigate the scalability of training Inception-v3 using multiple replicas. We configure TensorFlow with 7 PS tasks, and vary the number of worker tasks using two different clusters.

For the first experiment, we compare the performance training Inception using asynchronous SGD on TensorFlow and MXNet, a contemporary system using a parameter server architecture. For this experiment we use Google Compute Engine virtual machines running on Intel Xeon E5 servers with NVIDIA K80 GPUs, configured with 8 vCPUs, 16Gbps of network bandwidth, and one GPU per VM. Both systems use 7 PS tasks running on separate VMs with no GPU. Figure 8(a) shows that TensorFlow achieves performance that is marginally better than MXNet. As expected, the results are largely determined by single-GPU performance, and both systems use cuDNN version 5.1, so they have access to the same optimized GPU kernels.

Using a larger internal cluster (with NVIDIA K40 GPUs, and a shared datacenter network), we investigate the effect of coordination (§4.4) on training performance. Ideally, with efficient synchronous training, a model such as Inception-v3 will train in fewer steps, and converge to a higher accuracy than with asynchronous training [10]. Training throughput improves to 2,300 images per second as we increase the number of workers to 200, but with diminishing returns (Figure 8(b)). As we add more workers, the step time increases, because there is more contention on the PS tasks, both at the network interface and in the aggregation of updates. As expected, for all configurations, synchronous steps are longer than asynchronous steps, because all workers must wait for the slowest worker to catch up before starting the next step. While the median synchronous step is approximately 10% longer than an asynchronous step with the same workers, above the 90th percentile the synchronous performance degrades sharply, because stragglers disproportionately impact tail latency.

To mitigate tail latency, we add backup workers so that a step completes when the first m of n tasks produce gradients. Figure 8(c) shows the effect of adding backup workers to a 50-worker Inception training job. Each additional backup worker up to and including the fourth reduces the median step time, because the probability of a straggler affecting the step decreases. Adding a fifth backup worker slightly degrades performance, because the 51st worker (i.e., the first whose result is discarded) is more likely to be a non-straggler that generates more incoming traffic for the PS tasks. Figure 8(c) also plots the *normalized speedup* for each configuration, defined as $t(b)/t(0) \times 50/(50+b)$ (where $t(b)$ is the median step time with b backup workers), and which discounts the speedup by the fraction of additional resources consumed. Although adding 4 backup workers achieves the shortest overall step time (1.93 s), adding 3 achieves the highest normalized speedup (9.5%), and hence uses less aggregate GPU-time to reach the same quality.

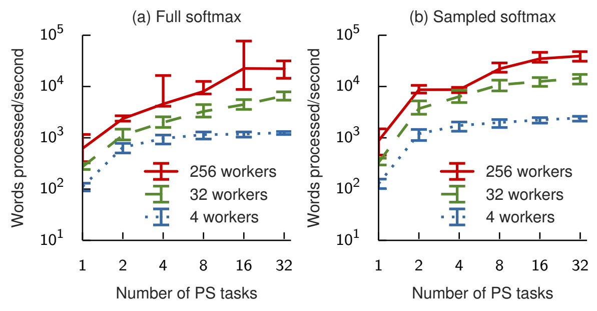


Figure 9: Increasing the number of PS tasks leads to increased throughput for language model training, by parallelizing the softmax computation. Sampled softmax increases throughput by performing less computation.

6.4 Language modeling

Given a sequence of words, a language model predicts the most probable next word [6]. Therefore, language models are integral to predictive text, speech recognition, and translation applications. In this experiment, we investigate how TensorFlow can train a recurrent neural network (viz. LSTM-512-512 [41]) to model the text in the One Billion Word Benchmark [9]. The vocabulary size $|V|$ limits the performance of training, because the final layer must decode the output state into probabilities for each of $|V|$ classes [37]. The resulting parameters can be large ($|V| \times d$ for output state dimension d) so we use the techniques for handling large models from Subsection 4.2. We use a restricted vocabulary of the most common 40,000 words—instead of the full 800,000 words [9]—in order to experiment with smaller configurations.

Figure 9 shows the training throughput, measured in words per second, for varying numbers of PS and worker tasks, and two softmax implementations. The *full* softmax (Figure 9(a)) multiplies each output by a 512 $\times$ 40,000 weight matrix sharded across the PS tasks. Adding more PS tasks increases the throughput, because TensorFlow can exploit distributed model parallelism [20, 43] and perform the multiplication and gradient calculation on the PS tasks, as in Project Adam [14]. Adding a second PS task is more effective than increasing from 4 to 32, or 32 to 256 workers. Eventually the throughput saturates, as the LSTM calculations dominate the training step.

The *sampled* softmax (Figure 9(b)) reduces the data transferred and the computation performed on the PS tasks [37]. Instead of a dense weight matrix, it multiplies the output by a random sparse matrix containing weights for the true class and a random sample of false classes. We sample 512 classes for each batch, thus reducing the softmax data transfer and computation by a factor of 78.

7 Conclusions

We have described the TensorFlow system and its programming model. TensorFlow's dataflow representation subsumes existing work on parameter server systems, and offers a set of uniform abstractions that allow users to harness large-scale heterogeneous systems, both for production tasks and for experimenting with new approaches. We have shown several examples of how the TensorFlow programming model facilitates experimentation (§4) and demonstrated that the resulting implementations are performant and scalable (§6).

Our initial experience with TensorFlow is encouraging. A large number of groups at Google have deployed TensorFlow in production, and TensorFlow is helping our research colleagues to make new advances in machine learning. Since we released TensorFlow as open-source software, more than 14,000 people have forked the source code repository, the binary distribution has been downloaded over one million times, and dozens of machine learning models that use TensorFlow have been published.

TensorFlow is a work in progress. Its flexible dataflow representation enables power users to achieve excellent performance, but we have not yet determined default policies that work well for all users. Further research on automatic optimization should bridge this gap. On the system level, we are actively developing algorithms for automatic placement, kernel fusion, memory management, and scheduling. While the current implementations of mutable state and fault tolerance suffice for applications with weak consistency requirements, we expect that some TensorFlow applications will require stronger consistency, and we are investigating how to build such policies at user-level. Finally, some users have begun to chafe at the limitations of a static dataflow graph, especially for algorithms like deep reinforcement learning [54]. Therefore, we face the intriguing problem of providing a system that transparently and efficiently uses distributed resources, even when the structure of the computation unfolds dynamically.

Acknowledgments

We gratefully acknowledge contributions from our colleagues within Google, and from members of the wider machine learning community. In particular, we appreciate the feedback we have received from the rest of the Google Brain team and the many users of DistBelief and TensorFlow. We thank the anonymous OSDI reviewers and our shepherd KyoungSoo Park for their suggestions, which greatly improved the presentation of this paper.

References

[1] M. Abadi, A. Agarwal, P. Barham, E. Brevdo, Z. Chen, C. Citro, G. S. Corrado, A. Davis, J. Dean, M. Devin, S. Ghemawat, I. J. Goodfellow, A. Harp, G. Irving, M. Isard, Y. Jia, R. Józefowicz, L. Kaiser, M. Kudlur, J. Levenberg, D. Mane, R. Monga, S. Moore, D. G. Murray, C. Olah, M. Schuster, J. Shlens, B. Steiner, I. Sutskever, K. Talwar, P. A. Tucker, V. Vanhoucke, V. Vasudevan, F. B. Viégas, O. Vinyals, P. Warden, M. Wattenberg, M. Wicke, Y. Yu, and X. Zheng. TensorFlow: Large-scale machine learning on heterogeneous distributed systems. *arXiv preprint*, 1603.04467, 2016. arxiv.org/abs/1603.04467. Software available from tensorflow.org.

[2] R. Al-Rfou, G. Alain, A. Almahairi, C. Angermueller, D. Bahdanau, N. Ballas, F. Bastien, J. Bayer, A. Belikov, A. Belopolsky, Y. Bengio, A. Bergeron, J. Bergstra, V. Bisson, J. Bleecher Snyder, N. Bouchard, N. Boulanger-Lewandowski, X. Bouthillier, A. de Brébisson, O. Breuleux, P.-L. Carrier, K. Cho, J. Chorowski, P. Christiano, T. Cooijmans, M.-A. Côté, M. Côté, A. Courville, Y. N. Dauphin, O. Delalleau, J. Demouth, G. Desjardins, S. Dieleman, L. Dinh, M. Ducoffe, V. Dumoulin, S. Ebrahimi Kahou, D. Erhan, Z. Fan, O. Firat, M. Germain, X. Glorot, I. Goodfellow, M. Graham, C. Gulcehre, P. Hamel, I. Harlouchet, J.-P. Heng, B. Hidasi, S. Honari, A. Jain, S. Jean, K. Jia, M. Korobov, V. Kulkarni, A. Lamb, P. Lamblin, E. Larsen, C. Laurent, S. Lee, S. Lefrancois, S. Lemieux, N. Léonard, Z. Lin, J. A. Livezey, C. Lorenz, J. Lowin, Q. Ma, P.-A. Manzagol, O. Mastropietro, R. T. McGibbon, R. Memisevic, B. van Merriënboer, V. Michalski, M. Mirza, A. Orlandi, C. Pal, R. Pascanu, M. Pezeshki, C. Raffel, D. Renshaw, M. Rocklin, A. Romero, M. Roth, P. Sadowski, J. Salvatier, F. Savard, J. Schlüter, J. Schulman, G. Schwartz, I. V. Serban, D. Serdyuk, S. Shabanian, E. Simon, S. Spieckermann, S. R. Subramanyam, J. Sygnowski, J. Tanguay, G. van Tulder, J. Turian, S. Urban, P. Vincent, F. Visin, H. de Vries, D. Warde-Farley, D. J. Webb, M. Willson, K. Xu, L. Xue, L. Yao, S. Zhang, and Y. Zhang. Theano: A Python framework for fast computation of mathematical expressions. *arXiv preprint*, 1605.02688, 2016. arxiv.org/abs/1605.02688.

[3] A. Angelova, A. Krizhevsky, and V. Vanhoucke. Pedestrian detection with a large-field-of-view deep network. In *Proceedings of ICRA*, pages 704–711. IEEE, 2015. www.vision.caltech.edu/anelia/publications/Angelova15LFOV.pdf.

[4] Arvind and D. E. Culler. Dataflow architectures. In *Annual Review of Computer Science Vol. 1, 1986*, pages 225–253. Annual Reviews Inc., 1986. www.dtic.mil/cgi-bin/GetTRDoc?Location=U2&doc=GetTRDoc.pdf&AD=ADA166235.

[5] J. Ba, V. Mnih, and K. Kavukcuoglu. Multiple object recognition with visual attention. *arXiv preprint*, 1412.7755, 2014. arxiv.org/abs/1412.7755.

[6] Y. Bengio, R. Ducharme, P. Vincent, and C. Jauvin. A neural probabilistic language model. *Journal of Machine Learning Research*, 3:1137–1155, 2003. jmlr.org/papers/volume3/bengio03a/bengio03a.pdf.

[7] T. Brants and A. Franz. Web 1T 5-gram version 1, 2006. catalog.ldc.upenn.edu/LDC2006T13.

[8] R. H. Byrd, G. M. Chin, J. Nocedal, and Y. Wu. Sample size selection in optimization methods for machine learning. *Mathematical Programming*, 134(1):127–155, 2012. dx.doi.org/10.1007/s10107-012-0572-5.

[9] C. Chelba, T. Mikolov, M. Schuster, Q. Ge, T. Brants, and P. Koehn. One billion word benchmark for measuring progress in statistical language modeling. *arXiv preprint*, 1312.3005, 2013. arxiv.org/abs/1312.3005.

[10] J. Chen, R. Monga, S. Bengio, and R. Jozefowicz. Revisiting distributed synchronous SGD. In *Proceedings of ICLR Workshop Track*, 2016. arxiv.org/abs/1604.00981.

[11] T. Chen, M. Li, Y. Li, M. Lin, N. Wang, M. Wang, T. Xiao, B. Xu, C. Zhang, and Z. Zhang. MXNet: A flexible and efficient machine learning library for heterogeneous distributed systems. In *Proceedings of LearningSys*, 2015. www.cs.cmu.edu/˜muli/file/mxnet-learning-sys.pdf.

[12] H.-T. Cheng, L. Koc, J. Harmsen, T. Shaked, T. Chandra, H. Aradhye, G. Anderson, G. Corrado, W. Chai, M. Ispir, R. Anil, Z. Haque, L. Hong, V. Jain, X. Liu, and H. Shah. Wide & deep learning for recommender systems. *arXiv preprint*, 1606.07792, 2016. arxiv.org/abs/1606.07792.

[13] S. Chetlur, C. Woolley, P. Vandermersch, J. Cohen, J. Tran, B. Catanzaro, and E. Shelhamer. cuDNN: Efficient primitives for deep learning. *arXiv preprint*, 1410.0759, 2014. arxiv.org/abs/1410.0759.

[14] T. Chilimbi, Y. Suzue, J. Apacible, and K. Kalyanaraman. Project Adam: Building an efficient and scalable deep learning training system. In *Proceedings of OSDI*, pages 571–582, 2014. www.usenix.org/system/files/conference/osdi14/osdi14-paper-chilimbi.pdf.

[15] S. Chintala. convnet-benchmarks, 2016. github.com/soumith/convnet-benchmarks.

[16] E. S. Chung, J. D. Davis, and J. Lee. LINQits: Big data on little clients. In *Proceedings of ISCA*, pages 261–272, 2013. www.microsoft.com/en-us/research/wp-content/uploads/2013/06/ISCA13_linqits.pdf.

[17] R. Collobert, S. Bengio, and J. Mariéthoz. Torch: A modular machine learning software library. Technical report, IDIAP, 2002. infoscience.epfl.ch/record/82802/files/rr02-46.pdf.

[18] H. Cui, H. Zhang, G. R. Ganger, P. B. Gibbons, and E. P. Xing. GeePS: Scalable deep learning on distributed GPUs with a GPU-specialized parameter server. In *Proceedings of EuroSys*, 2016. www.pdl.cmu.edu/PDL-FTP/CloudComputing/GeePS-cui-eurosys16.pdf.

[19] A. Dai, C. Olah, and Q. V. Le. Document embedding with paragraph vectors. *arXiv preprint*, 1507.07998, 2015. arxiv.org/abs/1507.07998.

[20] J. Dean, G. S. Corrado, R. Monga, K. Chen, M. Devin, Q. V. Le, M. Z. Mao, M. Ranzato, A. Senior, P. Tucker, K. Yang, and A. Y. Ng. Large scale distributed deep networks. In *Proceedings of NIPS*, pages 1232–1240, 2012. research.google.com/archive/large_deep_networks_nips2012.pdf.

[21] J. Dean and S. Ghemawat. MapReduce: Simplified data processing on large clusters. In *Proceedings of OSDI*, pages 137–149, 2004. research.google.com/archive/mapreduce-osdi04.pdf.

[22] DMLC. MXNet for deep learning, 2016. github.com/dmlc/mxnet.

[23] J. Duchi, E. Hazan, and Y. Singer. Adaptive subgradient methods for online learning and stochastic optimization. *Journal of Machine Learning Research*, 12:2121–2159, 2011. jmlr.org/papers/volume12/duchi11a/duchi11a.pdf.

[24] A. Frome, G. S. Corrado, J. Shlens, S. Bengio, J. Dean, T. Mikolov, et al. DeVISE: A deep visual-semantic embedding model. In *Proceedings of NIPS*, pages 2121–2129, 2013. research.google.com/pubs/archive/41473.pdf.

[25] J. Gonzalez-Dominguez, I. Lopez-Moreno, P. J. Moreno, and J. Gonzalez-Rodriguez. Frame-by-frame language identification in short utterances using deep neural networks. *Neural Networks*, 64:49–58, 2015. research.google.com/pubs/archive/42929.pdf.

[26] I. J. Goodfellow, J. Pouget-Abadie, M. Mirza, B. Xu, D. Warde-Farley, S. Ozair, A. C. Courville, and Y. Bengio. Generative adversarial nets. In *Proceedings of NIPS*, pages 2672–2680, 2014. papers.nips.cc/paper/5423-generative-adversarial-nets.pdf.

[27] Google Research. Tensorflow serving, 2016. tensorflow.github.io/serving/.

[28] K. He, X. Zhang, S. Ren, and J. Sun. Deep residual learning for image recognition. In *Proceedings of CVPR*, pages 770–778, 2016. arxiv.org/abs/1512.03385.

[29] G. Heigold, V. Vanhoucke, A. Senior, P. Nguyen, M. Ranzato, M. Devin, and J. Dean. Multilingual acoustic models using distributed deep neural networks. In *Proceedings of ICASSP*, pages 8619–8623, 2013. research.google.com/pubs/archive/40807.pdf.

[30] G. E. Hinton. Learning distributed representations of concepts. In *Proceedings of the Eighth Annual Conference of the Cognitive Science Society*, pages 1–12, 1986. www.cogsci.ucsd.edu/~ajyu/Teaching/Cogs202_sp13/Readings/hinton86.pdf.

[31] G. E. Hinton, L. Deng, D. Yu, G. E. Dahl, A. Mohamed, N. Jaitly, A. Senior, V. Vanhoucke, P. Nguyen, T. N. Sainath, and B. Kingsbury. Deep neural networks for acoustic modeling in speech recognition: The shared views of four research

groups. *IEEE Signal Process. Mag.*, 29(6):82–97, 2012. www.cs.toronto.edu/~gdahl/papers/deepSpeechReviewSPM2012.pdf.

[32] S. Hochreiter and J. Schmidhuber. Long short-term memory. *Neural computation*, 9(8):1735–1780, 1997. deeplearning.cs.cmu.edu/pdfs/Hochreiter97_lstm.pdf.

[33] S. Ioffe and C. Szegedy. Batch normalization: Accelerating deep network training by reducing internal covariate shift. In *Proceedings of ICML*, pages 448–456, 2015. jmlr.org/proceedings/papers/v37/ioffe15.pdf.

[34] M. Isard, M. Budiu, Y. Yu, A. Birrell, and D. Fetterly. Dryad: distributed data-parallel programs from sequential building blocks. In *Proceedings of EuroSys*, pages 59–72, 2007. www.microsoft.com/en-us/research/wp-content/uploads/2007/03/eurosys07.pdf.

[35] B. Jacob et al. gemmlowp: a small self-contained low-precision GEMM library, 2015. github.com/google/gemmlowp.

[36] B. Jacob, G. Guennebaud, et al. Eigen library for linear algebra. eigen.tuxfamily.org.

[37] S. Jean, K. Cho, R. Memisevic, and Y. Bengio. On using very large target vocabulary for neural machine translation. In *Proceedings of ACL-ICJNLP*, pages 1–10, July 2015. www.aclweb.org/anthology/P15-1001.

[38] Y. Jia, E. Shelhamer, J. Donahue, S. Karayev, J. Long, R. Girshick, S. Guadarrama, and T. Darrell. Caffe: Convolutional architecture for fast feature embedding. In *Proceedings of ACM Multimedia*, pages 675–678, 2014. arxiv.org/abs/1408.5093.

[39] M. I. Jordan. Serial order: A parallel distributed processing approach. ICS report 8608, Institute for Cognitive Science, UCSD, La Jolla, 1986. cseweb.ucsd.edu/~gary/PAPER-SUGGESTIONS/Jordan-TR-8604.pdf.

[40] N. Jouppi. Google supercharges machine learning tasks with TPU custom chip, 2016. cloudplatform.googleblog.com/2016/05/Google-supercharges-machine-learning-tasks-with-custom-chip.html.

[41] R. Józefowicz, O. Vinyals, M. Schuster, N. Shazeer, and Y. Wu. Exploring the limits of language modeling. *arXiv preprint*, 1602.02410, 2016. arxiv.org/abs/1602.02410.

[42] A. Karpathy, G. Toderici, S. Shetty, T. Leung, R. Sukthankar, and L. Fei-Fei. Large-scale video classification with convolutional neural networks. In *Proceedings of CVPR*, pages 1725–1732, 2014. research.google.com/pubs/archive/42455.pdf.

[43] A. Krizhevsky. One weird trick for parallelizing convolutional neural networks. *arXiv preprint*, 1404.5997, 2014. arxiv.org/abs/1404.5997.

[44] A. Krizhevsky, I. Sutskever, and G. E. Hinton. ImageNet classification with deep convolutional neural networks. In *Proceedings of NIPS*, pages 1106–1114, 2012. papers.nips.cc/paper/4824-imagenet-classification-with-deep-convolutional-neural-networks.pdf.

[45] H. Larochelle, Y. Bengio, J. Louradour, and P. Lamblin. Exploring strategies for training deep neural networks. *Journal of Machine Learning Research*, 10:1–40, 2009. jmlr.org/papers/volume10/larochelle09a/larochelle09a.pdf.

[46] A. Lavin and S. Gray. Fast algorithms for convolutional neural networks. In *Proceedings of CVPR*, pages 4013–4021, 2016. arxiv.org/abs/1509.09308.

[47] Q. Le, M. Ranzato, R. Monga, M. Devin, G. Corrado, K. Chen, J. Dean, and A. Ng. Building high-level features using large scale unsupervised learning. In *Proceedings of ICML*, pages 81–88, 2012. research.google.com/archive/unsupervised_icml2012.pdf.

[48] Y. LeCun, C. Cortes, and C. J. Burges. The MNIST database of handwritten digits, 1998. yann.lecun.com/exdb/mnist/.

[49] M. Li, D. G. Andersen, J. Park, A. J. Smola, A. Ahmed, V. Josifovski, J. Long, E. J. Shekita, and B.-Y. Su. Scaling distributed machine learning with the Parameter Server. In *Proceedings of OSDI*, pages 583–598, 2014. www.usenix.org/system/files/conference/osdi14/osdi14-paper-li_mu.pdf.

[50] C. J. Maddison, A. Huang, I. Sutskever, and D. Silver. Move evaluation in Go using deep convolutional neural networks. *arXiv preprint*, 1412.6564, 2014. arxiv.org/abs/1412.6564.

[51] F. McSherry, M. Isard, and D. G. Murray. Scalability! But at what COST? In *Proceedings of HotOS*, HOTOS'15, 2015. www.usenix.org/system/files/conference/hotos15/hotos15-paper-mcsherry.pdf.

[52] T. Mikolov, K. Chen, G. Corrado, and J. Dean. Efficient estimation of word representations in vector space. In *Proceedings of ICLR Workshops Track*, 2013. arxiv.org/abs/1301.3781.

[53] V. Mnih, N. Heess, A. Graves, and K. Kavukcuoglu. Recurrent models of visual attention. In *Proceedings of NIPS*, pages 2204–2212, 2014. papers.nips.cc/paper/5542-recurrent-models-of-visual-attention.pdf.

[54] V. Mnih, K. Kavukcuoglu, D. Silver, A. A. Rusu, J. Veness, M. G. Bellemare, A. Graves, M. Riedmiller, A. K. Fidjeland, G. Ostrovski, S. Petersen, C. Beattie, A. Sadik, I. Antonoglou, H. King, D. Kumaran, D. Wierstra, S. Legg, and D. Hassabis. Human-level control through deep reinforcement learning. *Nature*, 518(7540):529–533, 02 2015. dx.doi.org/10.1038/nature14236.

[55] P. Moritz, R. Nishihara, I. Stoica, and M. I. Jordan. SparkNet: Training deep networks in Spark. In *Proceedings of ICLR*, 2016. arxiv.org/abs/1511.06051.

[56] D. G. Murray, F. McSherry, M. Isard, R. Isaacs, P. Barham, and M. Abadi. Incremental, iterative data processing with timely dataflow. *Commun. ACM*, 59(10):75–83, Sept. 2016. dl.acm.org/citation.cfm?id=2983551.

[57] A. Nair, P. Srinivasan, S. Blackwell, C. Alcicek, R. Fearon, A. De Maria, V. Panneershelvam, M. Suleyman, C. Beattie, S. Petersen, et al. Massively parallel methods for deep reinforcement learning. *arXiv preprint*, 1507.04296, 2015. arxiv.org/abs/1507.04296.

[58] Nervana Systems. Neon deep learning framework, 2016. github.com/NervanaSystems/neon.

[59] NVIDIA Corporation. NCCL: Optimized primitives for collective multi-GPU communication, 2016. github.com/NVIDIA/nccl.

[60] R. Pascanu, T. Mikolov, and Y. Bengio. On the difficulty of training recurrent neural networks. In *Proceedings of ICML*, pages 1310–1318, 2013. jmlr.org/proceedings/papers/v28/pascanu13.pdf.

[61] B. Recht, C. Re, S. Wright, and F. Niu. Hogwild: A lock-free approach to parallelizing stochastic gradient descent. In *Proceedings of NIPS*, pages 693–701, 2011. papers.nips.cc/paper/4390-hogwild-a-lock-free-approach-to-parallelizing-stochastic-gradient-descent.pdf.

[62] C. J. Rossbach, Y. Yu, J. Currey, J.-P. Martin, and D. Fetterly. Dandelion: a compiler and runtime for heterogeneous systems. In *Proceedings of SOSP*, pages 49–68, 2013. sigops.org/sosp/sosp13/papers/p49-rossbach.pdf.

[63] D. E. Rumelhart, G. E. Hinton, and R. J. Williams. Learning representations by back-propagating errors. In *Cognitive modeling*, volume 5, pages 213–220. MIT Press, 1988. www.cs.toronto.edu/~hinton/absps/naturebp.pdf.

[64] O. Russakovsky, J. Deng, H. Su, J. Krause, S. Satheesh, S. Ma, Z. Huang, A. Karpathy, A. Khosla, M. Bernstein, A. C. Berg, and L. Fei-Fei. ImageNet Large Scale Visual Recognition Challenge. *International Journal of Computer Vision*, 115(3):211–252, 2015. arxiv.org/abs/1409.0575.

[65] A. Smola and S. Narayanamurthy. An architecture for parallel topic models. *Proc. VLDB Endow.*, 3(1–2):703–710, Sept. 2010. vldb.org/pvldb/vldb2010/papers/R63.pdf.

[66] I. Sutskever, J. Martens, G. E. Dahl, and G. E. Hinton. On the importance of initialization and momentum in deep learning. In *Proceedings of ICML*, pages 1139–1147, 2013. jmlr.org/proceedings/papers/v28/sutskever13.pdf.

[67] I. Sutskever, O. Vinyals, and Q. V. Le. Sequence to sequence learning with neural networks. In *Proceedings of NIPS*, pages 3104–3112, 2014. papers.nips.cc/paper/5346-sequence-to-sequence-learning-with-neural.pdf.

[68] C. Szegedy, W. Liu, Y. Jia, P. Sermanet, S. Reed, D. Anguelov, D. Erhan, V. Vanhoucke, and A. Rabinovich. Going deeper with convolutions. In *Proceedings of CVPR*, pages 1–9, 2015. arxiv.org/abs/1409.4842.

[69] C. Szegedy, V. Vanhoucke, S. Ioffe, J. Shlens, and Z. Wojna. Rethinking the Inception architecture for computer vision. *arXiv preprint*, 1512.00567, 2015. arxiv.org/abs/1512.00567.

[70] C. tao Chu, S. K. Kim, Y. an Lin, Y. Yu, G. Bradski, K. Olukotun, and A. Y. Ng. Map-reduce for machine learning on multicore. In *Proceedings of NIPS*, pages 281–288, 2007. papers.nips.cc/paper/3150-map-reduce-for-machine-learning-on-multicore.pdf.

[71] A. Verma, L. Pedrosa, M. Korupolu, D. Oppenheimer, E. Tune, and J. Wilkes. Large-scale cluster management at Google with Borg. In *Proceedings of EuroSys*, 2015. research.google.com/pubs/archive/43438.pdf.

[72] O. Vinyals, L. Kaiser, T. Koo, S. Petrov, I. Sutskever, and G. Hinton. Grammar as a foreign language. *arXiv preprint*, 2014. arxiv.org/abs/1412.7449.

[73] Y. Wu, M. Schuster, Z. Chen, Q. V. Le, M. Norouzi, W. Macherey, M. Krikun, Y. Cao, Q. Gao, K. Macherey, J. Klingner, A. Shah, M. Johnson, X. Liu, L. Kaiser, S. Gouws, Y. Kato, T. Kudo, H. Kazawa, K. Stevens, G. Kurian, N. Patil, W. Wang, C. Young, J. Smith, J. Riesa, A. Rudnick, O. Vinyals, G. Corrado, M. Hughes, and J. Dean. Google's Neural Machine Translation system: Bridging the gap between human and machine translation. *arXiv preprint*, 1609.08144, 2016. arxiv.org/abs/1609.08144.

[74] Y. Yu, M. Isard, D. Fetterly, M. Budiu, U. Erlingsson, P. K. Gunda, and J. Currey. DryadLINQ: A system for general-purpose distributed data-parallel computing using a high-level language. In *Proceedings of OSDI*, pages 1–14, 2008. www.usenix.org/legacy/event/osdi08/tech/full_papers/yu_y/yu_y.pdf.

[75] 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 NSDI*, pages 15–28, 2012. https://www.usenix.org/system/files/conference/nsdi12/nsdi12-final138.pdf.

[76] M. D. Zeiler, M. Ranzato, R. Monga, M. Mao, K. Yang, Q. Le, P. Nguyen, A. Senior, V. Vanhoucke, J. Dean, and G. E. Hinton. On rectified linear units for speech processing. In *Proceedings of ICASSP*, pages 3517–3521, 2013. research.google.com/pubs/archive/40811.pdf.

Exploring the Hidden Dimension in Graph Processing

Mingxing Zhang Yongwei Wu Kang Chen Xuehai Qian† Xue Li Weimin Zheng
*Tsinghua University** †*University of Southern California*

Abstract

Task partitioning of a graph-parallel system is traditionally considered equivalent to the graph partition problem. Such equivalence exists because the properties associated with each vertex/edge are normally considered indivisible. However, this assumption is not true for many Machine Learning and Data Mining (MLDM) problems: instead of a single value, a *vector* of data elements is defined as the property for each vertex/edge. This feature opens a new dimension for task partitioning because a vertex could be divided and assigned to different nodes.

To explore this new opportunity, this paper presents *3D partitioning*, a novel category of task partition algorithms that significantly reduces network traffic for certain MLDM applications. Based on 3D partitioning, we build a distributed graph engine CUBE. Our evaluation results show that CUBE outperforms state-of-the-art graph-parallel system PowerLyra by up to 4.7× (up to 7.3× speedup against PowerGraph).

1 Introduction

Efficient graph-parallel systems require careful task partitioning. It plays a pivotal role because the load balancing and communication cost are largely determined by the partitioning strategy. All existing partitioning algorithms in current systems assume that the property of each vertex/edge is indivisible. Therefore, task partitioning is equivalent to graph partitioning. But, in reality, the property associated with a(n) vertex/edge for many Machine Learning and Data Mining (MLDM) problems is a *vector* of data elements, which is *not* indivisible.

This new feature can be illustrated by a popular machine learning problem, Collaborative Filtering (CF), which estimates the missing ratings based on a given incomplete set of (user, item) ratings. The original problem is defined in a matrix-centric view: given a sparse rating matrix **R** with size $N \times M$, the goal is to find two dense matrices **P** (with size $N \times D$) and **Q** (with size $M \times D$) that are **R**'s non-negative factors (i.e., $\mathbf{R} \approx \mathbf{P} \times \mathbf{Q}^{\mathbf{T}}$). Here, N and M are the number of users and items, respectively. D is the size of feature vector. When formulated in a graph-centric view, the rows of **P** and **Q** correspond to

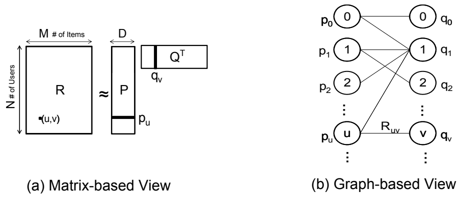


Figure 1: Collaborative Filtering.

vertices of a bipartite graph. Each vertex is associated with a property vector with D features. In contrast, the rating matrix **R** corresponds to edges. For every non-zero element (u, v) in matrix **R**, there is an edge connects vertex p_u and vertex q_v, and the weight of this edge is R_{uv}. An illustration of these two views is given in Figure 1.

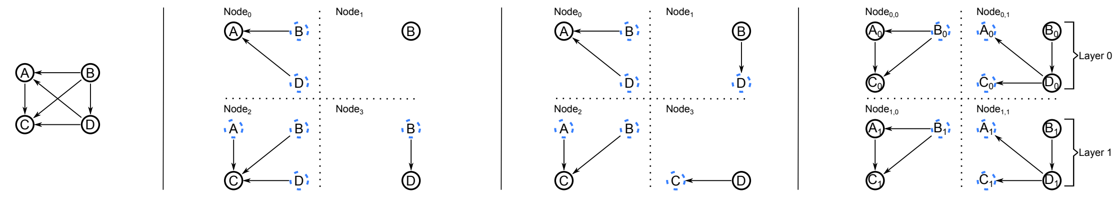
One distinct nature of the graph in Figure 1 (b) is that each vertex is associated with a divisible element vector, which is a common pattern when modelling MLDM algorithms as graph computing problems. Another good example is Sparse Matrix to Matrix Multiplication (SpMM), a prevalently used computation kernel that multiplies a dense feature matrix with a sparse parameter matrix (see Section 5.2.1 for more details). SpMM dominates the execution time of most minibatch-based neural network training algorithms.

In essence, when formulating matrix-based applications as graph problems, the property of vertex or edge is usually a vector of elements, instead of a single value. More importantly, during computation, these property vectors are mostly manipulated by *element-wise* operators, where the computations can be perfectly parallelized without any additional communication when disjoint ranges of vector elements are assigned to different nodes.

Due to the common pattern of vector property and its amenability to parallelism, this paper considers a *new dimension* of task partitioning, which is assigning disjoint elements of the same property to different nodes. It is considered to be a *hidden* dimension in existing 1D/2D partitioners used in previous systems [10, 15, 16, 24] because all of them treat the property as an indivisible component. According to our investigation, the *3D partitioning* principle could significantly reduce network traffic and improve performance.

The key intuition is that: since each node only processes a subset of elements in property vectors, it can be assigned with more edges and vertices that otherwise need to be assigned to different nodes. Therefore, on the bright side, certain communications previously happened

*M. Zhang, Y. Wu, K. Chen, X. Li and W. Zheng are with the Department of Computer Science and Technology, Tsinghua National Laboratory for Information Science and Technology (TNLIST), Tsinghua University, Beijing 100084, China; Technology Innovation Center at Yinzhou, Yangtze Delta Region Institute of Tsinghua University, Ningbo 315000, Zhejiang.



(a) Sample graph.

(b) 1-D partitioning: each vertex is attached with all its incoming edges.

(c) 2-D partitioning: the edges are equally partitioned.

(d) 3-D partitioning: each vertex is split into two sub-vertices, and a 2-D partitioner is used for each layer.

A B_0 : Vertices in blue dotted circles are replicas, while the others are masters.
It is a sub-vertex that contains only a subset of properties if a subscript is attach to the vertex's ID.

Figure 2: An illustration of 1D, 2D, and 3D partitioning.

between nodes are converted to local value exchanges. But, on the other side, 3D partitioning may incur extra synchronizations between sub-vertices/edges. In either case, with 3D partitioning, programmers are given the option to carefully choose the partition strategy of this third dimension. This ability enables them to explore a new tradeoff that may lead to better performance, which is prohibited by traditional 1D/2D partitioners. Importantly, 3D partitioning does not require long property vector to be effective. Our results show that a network traffic reduction up to 90.6% can be achieved by partitioning this dimension into just *64 layers*. In other words, our algorithm works very well on property vectors with modest and reasonable size.

Based on a novel 3D partitioning algorithm, we build a distributed graph processing engine CUBE, which introduces significantly fewer communication than existing systems in many real-world cases. To achieve better performance, CUBE internally uses a matrix-based data structure for storing and processing graphs while providing a set of vertex-centric APIs for the users. The matrix-based design is inspired by a recent graph-processing system [34], which only works on a *single* machine. The design of CUBE achieves both the programming productivity of vertex programming and the high performance of a matrix-based backend.

This paper makes the following contributions.

i) We propose the *first* 3D graph partitioning algorithm (Section 3.2) for graph-parallel systems. It considers a hidden dimension that is ignored by all previous systems. Unlike traditional 1D and 2D partitioning, the new dimension allows dividing the elements of property vectors to different nodes. Our 3D partitioning offers unprecedented performance that is not achievable by traditional graph partitioning strategies in existing systems.

ii) We propose a new programming model **UPPS** (**U**pdate, **P**ush, **P**ull, **S**ink) (Section 3.3) designed for 3D partitioning. The existing graph-oriented programming models are insufficient because they implicitly assume that the entire property of a single vertex is accessed as an indivisible component.

iii) We build CUBE, a graph processing engine that adopts 3D partitioning and implements the proposed vertex-centric programming model UPPS. The system significantly reduces communication cost and memory consumption. We use matrix-based data structures in the backend which reduces the COST metric [25] of our system to as low as four (Section 4).

iv) We systematically study the effectiveness of 3D partitioning with both micro-benchmarks (Section 5.2) and real-world MLDM algorithms (Section 5.3.3). The results show that it only trades a negligible growth of graph partitioning time for a notable reduction of both communication cost and memory consumption. Overall, CUBE outperforms state-of-the-art graph-parallel system PowerLyra by up to 4.7× (up to 7.3× speedup against PowerGraph).

2 Motivation and Background

An optimal task partitioning algorithm should *1)* ensure the balance of each node's computation load; and *2)* try to minimize the communication cost across multiple nodes. As the existing schemes assume that the property of each vertex is indivisible, the partitioning of graph-processing task is originally considered equivalent to graph partitioning. More specifically, existing partitioners try to optimally place the graph-structured data, including vertices and edges, across multiple machines, so that *1)* the number of edges on each node (which is roughly proportional to computation loads) is balanced; and *2)* the number of replicas (i.e., the number of shared vertices/edges which is proportional to the communication cost) is as small as possible. Two kinds of approaches exist for solving this problem: 1D partitioning and 2D partitioning.

1D Both GraphLab [22] and Pregel [23] adopt a **1D** partitioning algorithm. It assigns each node a disjoint set of vertices and all the connected incoming/outcoming edges. This algorithm is enough for randomly generated graphs, but for real-world graphs that follow the power law, a 1D partitioner usually leads to considerable skewness [15].

2D To avoid the drawbacks of 1D partitioning, recent systems [8, 15] are based on **2D** partitioning algorithms, in which the the graph is partitioned by edge rather than vertex. With a 2D partitioner, the edges of a graph will be equally assigned to each node. The system will set up replica of vertices to enable computation, and the automatic synchronization of these replicas requires communication. Various heuristics have been proposed to reduce communication cost by generating fewer number of replicas. For example, PowerLyra [10] uses a hybrid graph partitioning algorithm (named *Hybrid-cut*) that combines 1D partitioning and 2D partitioning with heuristics. By treating high-degree and low-degree vertices differently, Hybrid-cut achieves much lower communication cost on many real-world datasets. However, Hybrid-cut is still a special case of 2D partitioning that does not assign the same property vector to different nodes.

3D In many MLDM problems, a vector of data elements is associated to each vertex or edge hence the assumption of indivisible property is untrue and not necessary. This new dimension for task partitioning naturally leads to a new category of **3D** partitioning algorithms. To be more specific, for an N-node cluster a 3 partitioner will use L copies of the graph topology, where L is the number of layers and N is divisible by L. Each of these copies is partitioned by a regular 2D partitioner among a layer of only N/L nodes. On the other hand, the vector data associated to the graph are partitioned across layers evenly. In this setting, each layer occupies N/L nodes and the same graph with only subset of elements ($1/L$ of the original property vector) in its edges/vertices are partitioned among these N/L nodes by a regular 2D partitioner. Therefore, each vertex is split into L sub-vertices and the i^{th} layer maintains a copy of the graph that comprises of all the i^{th} sub-vertices/edges. 3D partitioning reduces communication cost **along edges** (e.g., synchronizations caused by element-wise operators), because the graph is partitioned across fewer nodes in each layer, thus each node in a layer could be assigned with more vertices and edges. This essentially converts the otherwise inter-node communication to local data exchanges.

Figure 2 compares the different partition algorithms applied on the graph in Figure 2 (a). In 1D partitioning (Figure 2 (b)), each node is assigned with one vertex and the incoming edges. There are six replicas in total. In 2D partitioning (Figure 2 (c)), edges are evenly partitioned, which leads to the same number of replicas as 1D partitioning.

Figure 2 (d) illustrates the concepts of 3D partitioning, where N is 4 and L is 2. First, the total of 4 nodes are divided to two layers. We denote each node as $Node_{i,j}$, where i is the layer index and j is the node index within a layer. Second, the graph is partitioned in the same way in both layers using a 2D partitioning algorithm. Different from 1D and 2D partitioning, since the number of nodes for each layer is halved, each node is assigned with more vertices and edges. In the example, the first node in all layers ($Node_{0,0}$ and $Node_{1,0}$) are assigned with 3 edges and 3 connected vertices, in which 1 vertex is replica. The second node in all layers ($Node_{0,1}$ and $Node_{1,1}$) are also assigned with 3 edges and 4 connected vertices, but among which 2 vertices are replicas. The increased number of vertices and edges in each node (3 edges in each layer of Figure 2 (d) compared to 1 or 2 edges in Figure 2 (b),(c)) translates to the reduced number of replicas needed for each layer (3 replicas in Figure 2 (d)) compared to 6 in Figure 2 (b),(c)). Although the total number of replicas (*3 replicas* $\times$ *2 layers = 6 replicas*) in all layers stays the same, the size of each replica is halved, therefore, the network traffic needed for replica synchronization is halved[1]. In essence, a 3D partitioning algorithm reduces the number of sub-graphs in each layer and hence reduces the **intra-layer** replica synchronization overhead.

However, 3D partitioning will incur a new kind of synchronization not needed before: the **inter-layer** synchronization between sub-vertices/edges. Therefore, programmers should carefully choose the number of layers to achieve the best performance. Nevertheless, the traditional 1D and 2D partitioning do not allow programmers to explore this tradeoff. A detailed discussion of this performance tradeoff is given in Section 5.

3 Programming Model

Existing graph-oriented programming models (e.g. GAS [15], TripletView [16], Pregel [23]) are designed for 1D/2D partitioning algorithms. They are insufficient for 3D partitioning because it is assumed that all elements of a property vector are accessed as an indivisible component. Thus, we adapt the popular GAS model and incorporate it with 3D partitioning, which leads to a new model named **UPPS** (**U**pdate, **P**ush, **P**ull, **S**ink) that accommodates 3D partitioning requirements. In this section, we first introduce UPPS and describe how a graph can be partitioned in the 3D fashion using UPPS. Then we explain the operations of UPPS and demonstrate their usages with two examples.

3.1 Data

As a vertex-centric model, UPPS models the user-defined data D as a directed data graph G, which consists of a set of vertices V together with a set of edges E. Users are allowed to associate arbitrary type of data with vertices and edges. The data attached to each vertex/edge are partitioned into two classes: *1)* an indivisible property *DShare* that is represented by a single variable; and *2)* a divisible collection of property vector elements *DColle*, which is stored as a **vector** of variables. The detailed specification of UPPS is given in Table 1.

[1] In some cases, there may be a shared part of every sub-vertices. We will discuss this situation later.

Table 1: The programming model UPPS.

Data					
G	—	$\{V, E, D = \{DShare, DColle\}, S_C\}$	$G_{bipartite}$	—	$\{\mathbb{U}, \mathbb{V}, E, D = \{DShare, DColle\}, S_C\}$
$DShare_u$	—	a single variable	$DShare_{u\to v}$	—	a single variable
$DColle_u$	—	a vector of variable with size S_C	$DColle_{u\to v}$	—	a vector of variable with size S_C
$DColle_u[i]$	—	the i^{th} element of $DColle_u$	$DColle_{u\to v}[i]$	—	the i^{th} element of $DColle_{u\to v}$
$D_u[i]$	—	abbreviation of $\{DShare_u, DColle_u[i]\}$	$D_{u\to v}[i]$	—	abbreviation of $\{DShare_{u\to v}, DColle_{u\to v}[i]\}$

Computation		
UpdateVertex($\mathcal{F}$)	—	**foreach** vertex $u \in V$ **do** $D_u^{new} := \mathcal{F}(D_u)$;
UpdateEdge($\mathcal{F}$)	—	**foreach** edge $(u,v) \in E$ **do** $D_{u\to v}^{new} := \mathcal{F}(D_{u\to v})$;
Push($\mathcal{G}$, $\mathcal{A}$, $\oplus$)	—	**foreach** vertex $v \in V$, index $i \in [0, S_C)$ **do** $DColle_v^{new}[i] := \mathcal{A}(D_v[i],\ \bigoplus_{(u,v)\in E}(\mathcal{G}(D_u[i],\ D_{u\to v}[i])))$;
Pull($\mathcal{G}$, $\mathcal{A}$, $\oplus$)	—	**foreach** vertex $u \in V$, index $i \in [0, S_C)$ **do** $DColle_u^{new}[i] := \mathcal{A}(D_u[i],\ \bigoplus_{(u,v)\in E}(\mathcal{G}(D_v[i],\ D_{u\to v}[i])))$;
Sink($\mathcal{H}$)	—	**foreach** edge $(u,v) \in E$, index $i \in [0, S_C)$ **do** $DColle_{u\to v}^{new}[i] := \mathcal{H}(D_u[i],\ D_v[i],\ Du\to v[i])$;

Users are required to assign an integer S_C as the **collection size** that defines the size of each *DColle* vector. When only *DShare* part of the edge data is used, *DColle* of edges can be set to *NULL*. But, if *DColle* of vertices and edges are both enabled, UPPS requires that their length should be equal. This restriction avoids inter-layer communication for certain operations (see Section 3.3). Moreover, if the input graph is undirected, the typical practice is using two directed edges (in each direction) to replace each of the original undirected edge. But, for many bipartite graph based MLDM algorithms, only one direction is needed (see more details in Section 3.6).

3.2 3D Partitioning

By explicitly decoupling the divisible property vector *DColle* and the indivisible part *DShare*, UPPS allows users to divide each vertex/edge into several sub-vertices/edges so that each of them has a copy of *DShare* and a **disjoint subset** of *DColle*. Based on UPPS, a 3D partitioner could be constructed by first dividing nodes into layers based on a layer count L and then partitioning the sub-graph in each layer following a 2D partitioning algorithm $\mathcal{P}$. Thus, a 3D partitioner can be denoted as $(\mathcal{P}, L)$.

Specifically, we should first guarantee that N is divisible by L. Then, the partitioner will *1)* equally group the nodes into L layers so that each layer contains N/L nodes; *2)* partition edge set E into N/L sub-sets with the 2D partitioner $\mathcal{P}$; and *3)* randomly separate vertex set V into N/L sub-sets. In the rest of this paper, we use $Node_{i,j}$ to denote the j^{th} node of the i^{th} layer; E_j and V_j to denote the j^{th} subset of E and V, respectively.

With the above definitions, after partition, $Node_{i,j}$ contains the following data copies:

- a shared copy of $DShare_u$, if vertex $u \in V_j$;
- an exclusive copy of $DColle_u[k]$, if vertex $u \in V_j$ and $LowerBound(i) \le k < LowerBound(i+1)$;
- a shared copy of $DShare_{u\to v}$, if edge $(u,v) \in E_j$;
- an exclusive copy of $DColle_{u\to v}[k]$, if edge $(u,v) \in E_j$ and $LowerBound(i) \le k < LowerBound(i+1)$;

In the above equations, $LowerBound(i)$ equals to $i * (\lfloor S_C/L \rfloor) + min(i, S_C\%L)$. In other words, each layer of the nodes contains a shared copy of all the *DShare* data and an exclusive sub-set of the *DColle* data.

In a 3D partitioning $(\mathcal{P}, L)$, both L and $\mathcal{P}$ affect the communication cost. When $L = N$, each layer only has one node which keeps the entire graph and processes $1/L$ of *DColle* elements. In this case, no replica for *DColle* data is needed, and the intra-layer communication cost is zero. But, it could potentially incur higher inter-layer communication due to synchronization between sub-vertices/edges. When $L = 1$, there is only one layer and $(\mathcal{P}, L)$ is degenerated to the 2D partitioning $\mathcal{P}$. Therefore, the communication cost is purely determined by $\mathcal{P}$. The common practice is to choose the L between 1 and N, so that both L and $\mathcal{P}$ will affect communication cost. The programmers are responsible for investigating the tradeoff and choosing the best setting. To help users choose the appropriate L, we provide the equations to calculate communication costs for different UPPS operations which can be used as building blocks for real applications (see Section 5.2). Within a layer, one can choose any 2D partitioning $\mathcal{P}$ and it is orthogonal to L.

3.3 Computation

UPPS has four types of operations which resemble the name of the model: **U**pdate, **P**ush, **P**ull, and **S**ink. The definition of these operations are given in Table 1. All possible variant forms of computations allowed in UPPS are also encoded in these APIs.

Update This operation takes **all** the information of each vertex/edge to calculate the new value. Roughly, *Update* operates on all elements of an edge or vertex in *vertical* direction. Since vertices and edges may be split into sub-vertices/edges, each node $Node_{i,j}$ needs to synchronize with nodes in other layers while updating. Note that *Update* only incurs inter-layer communicate between a node and nodes in other layers that share the same subset of vertices (V_j) or edges (E_j) (i.e., $Node_{*,j}$).

Push, Pull, Sink These three operations handles updates in *horizontal* direction: the updates follow the dependency relations determined by graph structure. For each edge $(u,v) \in E$: *Push* operation uses data of vertex u and edge (u,v) to update vertex v; *Pull* operation uses data of vertex v and edge (u,v) to update vertex u; *Sink* operation uses data of u and v to update the edge (u,v).

Push/Pull operation resembles the popular GAS (Gather, Apply, Scatter) operation. In GAS, each vertex reads data from its in-edges with the gather function $\mathcal{G}$, generates the updated value based on sum function $\oplus$, which is used to update the vertex using the apply function $\mathcal{A}$. UPPS further partitions property vertex, which is always considered as an indivisible component in GAS. To avoid inter-layer communication, UPPS restricts that the i^{th} *DColle* element of each vertex/edge will only depend on either *DShare* (which is by definition replicated in *all* layers) or the i^{th} *DColle* element of other vertices/edges (which is by definition exist in the *same* layer). Similar restriction applies to *Sink*. In other words, $Node_{i,j}$ only communicates to $Node_{i,*}$ in *Push/Pull/Sink*.

3.4 Bipartite Graph

Many MLDM problems model their input graphs as bipartite graphs, where vertices are separated into two disjoint sets $\mathbb{U}$ and $\mathbb{V}$ and edges connect pairs of vertices from $\mathbb{U}$ and $\mathbb{V}$. A recent study [11] demonstrates the unique properties of bipartite graphs and the special need of differentiated processing for vertices in $\mathbb{U}$ and $\mathbb{V}$. To capture this requirement, UPPS provides two additional APIs: *UpdateVertexU* and *UpdateVertexV*. They only update the vertices in $\mathbb{U}$ and $\mathbb{V}$, respectively. We use the bipartite-specialized 2D partitioner *bi-cut* [11] as $\mathcal{P}$ for bipartite graphs.

3.5 Compare with GAS

UPPS also follows the popular "think as a vertex" philosophy so that it is easy for programmers to use. In fact, the popular GAS model is a special case of UPPS that has $S_C \leq 1$[2]. Thus, users only need to make moderate changes to their original programs if they just want to take advantage of our efficient matrix backend.

[2]In this case, the workers can only be partitioned into one layer and hence our 3D partitioner degenerates to a traditional 1D/2D partitioner.

In contrast, if the users want to reduce the communication cost by using a 3D partitioner, the workers should be partitioned into at least two layers. As we will show, many popular algorithms can benefit from 3D partitioning without significant program change. Take the breadth-first search (BFS) as an example, in GAS, it can be implemented by: *1)* associating a boolean property to each vertex, which represents whether this vertex has been accessed or not; *2)* propagating this property in the Scatter phase; and *3)* using boolean 'OR' operation in both the Gather and Apply phase. In order to extend this application to do multi-source BFS, users of GAS model can simply *1)* replacing the original boolean variable of each vertex to a vector of boolean variables with length k, where k is the number of sources; and *2)* using element-wise 'OR' operation in the Gather and Apply phase. We see that the computation of GAS-based multi-source BFS is dominated by element-wise operations of two vectors, which shows a notable sign of optimization opportunity with 3D partitioning and UPPS. In fact, with UPPS, the multi-source BFS can be simply implemented by using the **same** program as original BFS. The only difference is that the collection size S_C is set to k rather than one.

Moreover, although it is not used in the above example, users of UPPS may want to have a complete view of the whole vector property of vertices/edges. In 3D partitioning, this intention results in a new kind of inter-layer communication, which inevitably leads to additional APIs (our Update operations). Examples of the usages of these new APIs are given in the next section.

Algorithm 1 Program for GD.

```
Data:
    $S_C$ :— $D$
    $DShare_u$ :— NULL;  $DShare_{u \to v}$ :— {double $Rate$, double $Err$}
    $DColle_u$, $DColle_{u \to v}$ :— vector<double>($S_C$)
Functions:
    $F_1(u_i, v_i, e_i)$ :— {return $u_i.DColle[i] * v_i.DColle[i]$;}
    $F_2(e)$ :— {
        $e.DShare.Err := sum(e.DColle) - e.DShare.Rate$;
        return $e$;
    }
    $F_3(u_i, e_i)$ :— {return $e_i.DShare.Err * u_i.DColle[i]$; }
    $F_4(v_i, \Sigma)$ :— {return $v_i.DColle[i] + \alpha * (\Sigma - \alpha * v_i.DColle[i])$;}
Computation for each iteration:
    $Sink(F_1)$;
    $UpdateEdge(F_2)$;
    $Pull(F_3, F_4, +)$;
    $Push(F_3, F_4, +)$;
```

3.6 Examples

For showcasing the usages of UPPS, we implemented two different algorithms that both solve the Collaborative filtering (CF) problem. The two algorithms together cover the usage of all operations in UPPS. In this section, we only explain at a high-level what UPPS operations do. The detailed implementation of each UPPS operation is given in Section 4.

CF is a kind of problems that estimate the missing ratings based on a given incomplete set of (user, item) ratings. Specifically, if we use N to denote the number of users and M to denote the number of items, input of CF is $R = \{R_{u,v}\}_{N\times M}$, which is a sparse user-item matrix where each item $R_{u,v}$ represents the rating of item v given from user u. The output of CF is two matrices P and Q, which are the user feature matrix and item feature matrix, respectively. P_u and Q_v are feature vectors of user u and item v, and each of them has a size of D. If we use $Err_{u,v}$ to represent the current prediction error of user-item pair (u,v), it is calculated by subtracting the dot product of the corresponding feature vectors with the actual rate, i.e., $Err_{u,v} = <P_u,\ Q_v^T> - R_{u,v}$. The object function of CF is minimizing $\sum_{(u,v)\in R} Err_{u,v}^2$.

GD Gradient Descent (GD) algorithm [20] is a classical solution to solve CF problem, which involves randomly initializing feature vectors and improving them iteratively. The parameters of this algorithm are updated by a magnitude proportional to the learning rate α in the opposite direction of the gradient, which results in the following update rules:

$$P_i^{new} := P_i + \alpha * (Err_{i,j} * Q_j - \alpha * P_i)$$
$$Q_j^{new} := Q_j + \alpha * (Err_{i,j} * P_i - \alpha * Q_j)$$

The program of GD implemented in UPPS is given by Algorithm 1, in which + is an abbreviation of the simple "sum" function. For simplicity, we do not show regularization code used to impose non-negativity on P and Q. In Algorithm 1, the collection size S_C is set to D, hence each vertex/edge's *DColle* part is a vector of *double* with length D. For vertices, which are used for modeling the users and items, these vectors are used to store the corresponding feature vector of the user/item. For edges, these vectors are temporary buffers for reserving partial results of the dot production. As for shared data, the *DShare* part of each edge (u,v) is a pair of $\{$*double* *Rate*, *double* *Err*$\}$, which represents the rating given to item v from user u and the current error in predicting this rating. In contrast, *DShare* for vertices are not used.

With the data defined as above, the computation of Algorithm 1 is almost an one-to-one translation of the above equations. In the first step, Algorithm 1 calculates prediction errors $Err_{u,v}$ for every given rating (i.e., every edge) by: *1)* using a *Sink* operation to compute the production of every aligned feature elements and store the result in the edge's *DColle* vector; and *2)* using an *UpdateEdge* operation to sum up each edge's *DColle* vector (i.e., $<P_u,\ Q_v^T>$) and subtract it with the corresponding *Rate*. After calculating the current errors, the updating formulas mentioned above can be implemented in a straightforward way (the *Pull* and the *Push* operation in Algorithm 1).

As shown in Algorithm 1, programmers only need to define the *Push*, *Pull* and *Sink* operation on *one* element of the *DColle* vector (i.e., the user-defined functions operate only one index i), while the *UpdateEdge* and *UpdateVertex* reads or writes all vector elements. Importantly, programmers do *not* need to specify "which sub-vertex/edge contains which *DColle* elements". The details such as indexes of data elements for each layer are specified in a decoupled manner and automatically handled by the framework (Section 4.4).

ALS Alternating Least Squares (ALS) [38] is another algorithm to solve CF problem. It alternatively fixes one unknown feature matrix and solves another by minimizing the object function $\sum_{(u,v)\in R} Err_{u,v}^2$. This approach turns a non-convex problem into a quadratic one that can be solved optimally. A general description of ALS is as follows:

Step 1 Randomly initialize matrix P.

Step 2 Fix P, calculate the best Q that minimizes the error function. This can be implemented by setting $Q_v = (\sum_{(u,v)\in R} P_u^T P_u)^{-1} (\sum_{(u,v)\in R} R_{u,v} P_u^T)$.

Step 3 Fix Q, calculate the best P in a similar way.

Step 4 Repeat Steps 2 and 3 until convergence.

As a typical bipartite algorithm, we implement ALS with the specialized APIs described in Section 3.4. Algorithm 2 presents our program, where the regularization code is also omitted. In ALS, the collection size S_C is set to "$D + D * D$" rather than just D. Each of the *DColle* vector contains two parts: *1)* a feature vector *Vec* with size D that stores the corresponding feature vector; and *2)* a buffer *Mat* with size $D\times D$, which is used to keep the result of $Vec^T * Vec$.

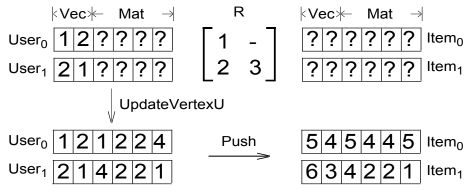


Figure 3: An illustration of ALS's Step 2. In this example, there are two users, two items, and three given ratings.

Figure 3 presents a typical example of ALS' Step 2. First, only the users' feature matrix P is initialized, so that only the feature vector of every vertex in $\mathbb{U}$ contains valid data; and the others are '?'. Then, an *UpdateVertexU* operation is used to calculate $Vec^T * Vec$ for every vertex in $\mathbb{U}$, and the results are stored in the corresponding *Mat* area. After that, a *Push* operation is used to aggregate the corresponding values. For each $v \in \mathbb{V}$, $\sum_{(u,v)\in R} R_{u,v} P_u^T$ and $\sum_{(u,v)\in R} P_u^T P_u$ are calculated and stored in v's *Vec* (i.e., *DColle[0:D-1]*) and *Mat* (i.e., *DColle[D:D+D*^{2}*-1]*) area, respectively. Finally, the optimal value of Q_v is calculated by solving a linear equation, which is implemented by calling the DSYSV function in LAPACK [2] (not illustrated in Figure 3). Similarly to Step 2, Step 3 of ALS can be implemented with the symmetrical call of *UpdateVertexV*, *Pull*, and *UpdateVertexU*.

Algorithm 2 Program for ALS.

```
Data:
    $S_C$ :— $D + D * D$
    $DShare_u$ :— NULL;    $DShare_{u \to v}$ :— {double Rate}
    $DColle_u$ :— vector<double>($S_C$);    $DColle_{u \to v}$ :— NULL
Functions:
    $F_1(v)$ :— {
        foreach $(i, j)$ from $(0,0)$ to $(D-1, D-1)$ do
            $v.DColle[D + i * D + j] := v.DColle[i] * v.DColle[j]$;
        return $v$;
    }
    $F_2(u_i, e_i)$ :— {
        if $i < D$ do return $e_i.DShare.Rate * u_i.DColle[i]$;
        else return $u_i.DColle[i]$;
    }
    $F_3(v)$ :— { DSYSV($D, \&v.DColle[0], \&v.DColle[D]$); return $v$;}
Computation for each iteration:
    UpdateVertexU($F_1$);
    Push($F_2$, +, +);
    UpdateVertexV($F_3$);
    UpdateVertexV($F_1$);
    Pull($F_2$, +, +);
    UpdateVertexU($F_3$);
```

4 Cube

To implement UPPS model, we build a new graph computing engine, Cube. It is written in C++ and based on MPICH2.

4.1 Graph Loading and Partitioning

In Cube, each node starts by loading a separate subset of the graph. The 3D partitioning algorithm in Cube consists of a 2D partitioning algorithm $\mathcal{P}$ and a layer count L, in which L is assigned by users. Thus, after loading, the 2D partitioner $\mathcal{P}$ is used to calculate an assignment of edges (i.e., E_j defined in Section 3.2); and, similarly, a random partitioner is used to partitioning the vertices (i.e., V_j). With these assignments, a global shuffling phase is followed to dispatch the loaded data to where they should be according to the partition policy given in Section 3.2. After shuffling, each $Node_{i,j}$ contains a copy of $D_a[k]$ and $D_{b \to c}[k]$, if vertex $a \in V_j$, edge $(b \to c) \in E_j$, and $LowerBound(i) \leq k < LowerBound(i+1)$. Moreover, we use *Hybrid-cut* [10] as the default 2D partitioner and *Bi-cut* [11] is used for bipartite graphs, as they work well on real-world graphs.

4.2 Update

In an *Update*, all the elements of *DColle* properties are needed. To implement this kind of operation, each vertex or edge is assigned a node as the master to perform the *Update*, which needs to gather all the required data before execution. The master node then iterates all data elements it collected, applies the user-defined function and finally scatters the updated values. For bipartite graph oriented operations, *UpdateVertexU* and *UpdateVertexV*, only a subset of vertex data is gathered.

As defined before, E_j and V_j are the subset of edges and vertices in j^{th} partition determined by a 2D partitioning algorithm, and $Node_{*,j}$ is the set of nodes in all layers to process E_j and V_j. In *Update*, each edge or vertex in E_j (or V_j) should have *one* master node $Node_{i,j}$, $i \in [0, L)$ among $Node_{*,j}$ that needs to gather all data elements for the edge or vertex to perform update operation. We define the set of edges or vertices of which the master node is $Node_{i,j}$ as $E_{i,j}$ or $V_{i,j}$. So we have $\bigcup_{i=0}^{L-1} E_{i,j} = E_j$ and $\bigcup_{i=0}^{L-1} V_{i,j} = V_j$. For simplicity, we randomly select a node from $Node_{*,j}$ for each edge and vertex in E_j and V_j. The inter-layer communications are incurred in *Update* by gathering and scattering, which are implemented by two rounds of *AllToAll* communication among the same nodes in different layers (i.e. $Node_{*,j}$).

For certain associative operations (e.g. sum), only the aggregation of the elements in a node is needed. For example, GD algorithm (Algorithm 1) only requires the sum of each node's local *DColle* elements. We allow users to define a **local combiner** for *Update* operations. With the local combiner, each node reduces its local *DColle* elements before sending the single value to its master. Local combiner further reduces communication because the master node only needs to gather one rather than S_C/L elements from each node in all other layers. For operations that can be specified by a custom *MPI_OP*, we leverage the existing *MPI_AllReduce* operation instead of gather and scatter to further reduce network traffic.

4.3 Push, Pull, Sink

A replica for $D_u[i]$ exists at node $Node_{i,j}$ if $\exists v : (u,v) \in E_j$ or $\exists v : (v,u) \in E_j$. The execution of each operation starts with replica synchronization within each layer. It could be implemented by executing L *AllToAll* communications among $Node_{i,*}$ concurrently in each layers.

After synchronization, for *Push* and *Pull*, the user-defined gather function $\mathcal{G}$ is used to calculate the gather result for each vertex; for *Sink*, the user defined function $\mathcal{H}$ is applied to each edge. After that, for *Push* or *Pull*, another L *AllToAll* communications among $Node_{i,*}$ are used to gather the results reduced by the user defined sum function $\oplus$ and then the user defined function $\mathcal{A}$ updates the vertex data. Similar to the *Update*, the sum function $\oplus$ is also used as a local combiner, so that the gather results are locally aggregated before sending. In bipartite mode, only a subset of vertex data is synchronized in *Push* and *Pull*.

Table 2: A collection of real-world graphs.

Dataset	$\lvert\mathbb{U}\rvert$	$\lvert\mathbb{V}\rvert$	$\lvert E\rvert$	Best 2D Partitioner	Description
Libimseti	135,359	168,791	17,359,346	Hybrid-cut	Dating data from libimseti.cz. [7]
Last.fm	359,349	211,067	17,559,530	Bi-cut	Music data from Last.fm. [9]
Netflix	17,770	480,189	100,480,507	Bi-cut	Movie review data from Netflix. [38]

4.4 Matrix-based data structure

Vertex-centric programming models are productive for developing graph programs. But, according to recent investigations, the performance of a naive vertex-cenric implementation can be $2\times - 6\times$ lower than matrix-based execution engines [19, 34, 37]. Therefore, CUBE uses matrix-based backend data structures.

In CUBE, both edge and vertex data are stored continuously. The edges are modeled as a sparse matrix and stored in coordinate list (COO) format, which is a list of (source vertex ID, destination vertex ID, edge value) tuples. The vertex data are simply stored in a continuous array. Since each worker only maintains a subset of graph data, the global ID of its vertices may not be continuous (e.g., vertex C is missing in sub-graph 1 of Figure 4). Thus we need to implement an efficient mechanism for index conversion. Many traditional graph engines use the inefficient hash map based data structure for indexing. Instead, CUBE maps the non-continuous global ID to continuous local ID for each worker. The mechanism is shown in Figure 4. In each worker, the vertex data are stored in a dense vector indexed by its local ID; the row/column ID of its sub-graph is substituted by local ID to ensure quick and straightforward location of the corresponding vertex data for each edge. Moreover, rather than using the simple dictionary order, we sort the edge data in **Hilbert order** [6], which is akin to ordered edges (a,b) by the interleaving of the bits of a and b. It has been shown that Hilbert order exhibits locality in both dimensions rather than one dimension as in the dictionary order, and hence incurs much fewer cache misses. Note that all the mapping and sorting procedures are performed in the initial preparing stage before the following many computing iterations. Therefore, the cost of the preparing procedure is amortized. The system records all data exchanging information at the preparing phase, so there is no need for global/local ID converting during the computation.

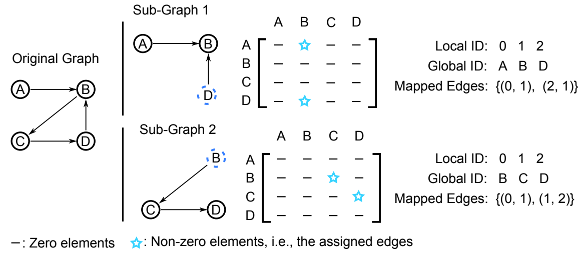


Figure 4: An illustration of the mapping between local and global IDs.

5 Evaluation

This section presents evaluation results of CUBE and compares it with two existing frameworks, PowerGraph [15] and PowerLyra [10]. For each case, we provide: *1)*. Mathematical equations that calculate the communication traffic; *2)*. Experimental performance results that validate the prediction based on communication traffic. To get a thorough understanding of CUBE, we also discuss other aspects such as scalability, memory consumption, partitioning cost, and COST metric.

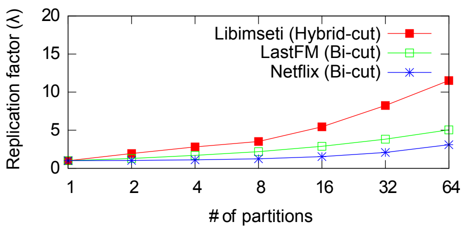
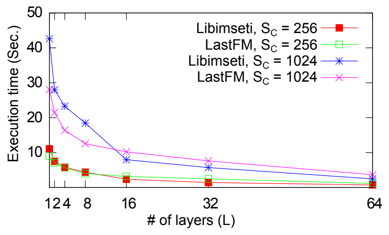


Figure 5: The best replication factor of each dataset.

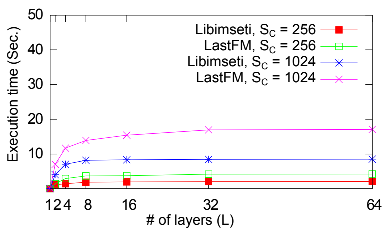
5.1 Evaluation setup

We conduct the experiments on an 8-node Intel® Xeon® CPU E5-2640 based system. All nodes are connected with a 1Gb Ethernet and each node has 8 cores running at 2.50 GHz. We use a collection of real-world bipartite graphs gathered by the Stanford Network Analysis Project [1]. Table 2 shows the basic characteristics of each dataset.

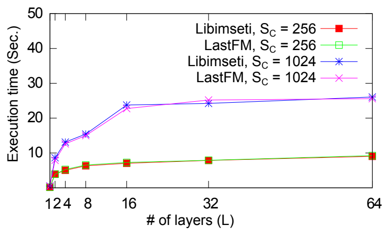
Since our 3D partitioning algorithm relies on a 2D partitioner within each layer, we first select the *best* 2D partitioner for each dataset. To do so, we evaluated **all** existing 2D partitioning algorithms in PowerGraph and PowerLyra. This includes the heuristic-based Hybrid-cut [10], the bipartite-graph-oriented algorithm Bi-cut [11] and many other random/hash partitioning algorithms. We calculated the average number of replicas for a vertex (i.e., replication factor, λ) for each algorithm. λ includes *both* original vertices and the replicas. We consider the best partitioner as the one that has the smallest λ. To capture the number of partitions, we use λ_x to denote the average number of replicas for a vertex when a graph is partitioned into x sub-graphs (e.g., $\lambda_1 = 1$). Table 2 also shows the best 2D partitioner for each data set: Hybrid-cut is the best for Libimseti, while Bi-cut is the best for LastFM and Netflix. For LastFM, source set (i.e., $\mathbb{U}$) should be used as the favorite subset, while for Netflix, target set (i.e., $\mathbb{V}$) should be used as the favorite subset. Here, "favorite subset" is an input parameter defined by Bi-cut that usually should be set to the larger vertex set of the bipartite graph.



(a) SpMM



(b) SumV



(c) SumE

Figure 6: The impact of layer count on average execution time for running the micro benchmarks with 64 workers.

Figure 5 shows the replication factor of each dataset for the best 2D partitioning algorithm. We see that Bi-cut is effective if the size of two vertex subsets in a bipartite graph is significantly skewed. It is indeed the case for Netflix: the size of its target subset (i.e., $|\mathbb{V}|$) is 27 times more than its source subset (i.e., $|\mathbb{U}|$). Therefore, the replication factor grows moderately with the number of partitions (e.g., λ_{64} of Netflix is only 3.09). On the other side, LastFM is more balanced and its replication factor grows faster. We show in later sections that, the faster the replication factor grows the better speedup our 3D partitioning algorithm can achieve. Therefore, the improvement is the most significant for Libimseti and the least for Netflix.

5.2 Micro Benchmarks

CUBE allows users to specify the layer count L which is a key factor determining the tradeoff between the amount of intra-layer and inter-layer communication. Two extreme values for L are: 1, where the inter-layer communication is zero and 3D partition degenerates to 2D partitioning; and N (the number of workers), where the intra-layer communication is zero. In general, as L becomes larger, the intra-layer communication decreases and inter-layer communication increases.

We present the equations to calculate communication traffic for three micro-benchmarks and show the performance results as L changes from 1 to 64. The reason why we use micro-benchmarks first before discussing full applications is two-fold. First, each micro-benchmark only requires a *single* operation in UPPS so that we can isolate it from other impacts. Second, the equations obtained for each case can be used as building blocks to construct communication traffic equations for real applications. We will show that the performance results can be indeed explained by the traffic equations.

5.2.1 SpMM

The Sparse Matrix to Matrix Multiplication (SpMM) multiplies a dense and small matrix **A** (size $D \times H$) with a big but sparse matrix **B** (size $H \times W$), where $D \ll H$, $D \ll W$. This computation kernel is prevalently used in many MLDM algorithms. For example, in training phase of some certain kinds of Deep Learning algorithms [14], the big sparse matrix **B** is used to represent the network parameters and the small dense **A** is a minibatch of training data, in which D is the batch size (usually ranges from 20 to 1000).

In UPPS, this problem could be modeled by a bipartite graph with $|V| = H + W$, where $|\mathbb{U}| = H$ and $|\mathbb{V}| = W$. The non-zero elements in the big sparse matrix are represented by an edge $i \rightarrow j$ (from a vertex in $\mathbb{U}$ to a vertex in $\mathbb{V}$) with $DShare_{i\rightarrow j} = b_{i,j}$ and $DColle_{i\rightarrow j} = NULL$. On the other side, the dense matrix **A** is modeled by vertices: the i^{th} column of **A** is represented as the *DColle* vector associated with vertex i in $\mathbb{U}$, where $S_C = D$ and $DShare = NULL$. The computation of a SpMM operation is implemented by a single *Push* (or *Pull*) operation.

Figure 6a shows the execution time of SpMM on 64 workers with L from 1 to 64. Since the computation of SpMM is always equally partitioned into each node, the reduction on execution time is mainly caused by the reduction on network traffic. Formally, if a 3D partitioner $(\mathcal{P}, L)$ is used for partitioning the graph into N nodes, a total of $\lambda_{N/L} * |V|$ replicas will be used in each layer. Since the communication of each Push/Pull operation only involves intra-layer communication and only the *DColle* elements of vertices are needed to be synchronized, the total network traffic can be calculated by summing the number of *DColle* elements sent in each layer, which is $(S_C/L) * (\lambda_{N/L} - 1) * |V|$.

For the bipartite graph in SpMM, synchronization is only needed among replicas in the sub-graph where the vertices are updated ($\mathbb{U}$ or $\mathbb{V}$). If SpMM is implemented as a *Push*, the network traffic is $(S_C/L) * (\lambda^{\mathbb{V}}_{N/L} - 1) * |\mathbb{V}|$; if it is implemented as a *Pull*, the network traffic is $(S_C/L) * (\lambda^{\mathbb{U}}_{N/L} - 1) * |\mathbb{U}|$. Here $\lambda^{\mathbb{U}}_{N/L}$ and $\lambda^{\mathbb{V}}_{N/L}$ are replication factor for $\mathbb{U}$ and $\mathbb{V}$, respectively.

As a result, the amount of network traffic in a SpMM operation can be calculated by the following equations, in which S denotes the size of each $DColle_u[i]$. The traffic is doubled because two rounds of communications (gather and scatter) are needed in replica synchronization.

$$\text{Traffic}(\text{SpMM}_{\text{Push}}) = 2 * \text{S} * S_C * (\lambda^{\mathbb{V}}_{N/L} - 1) * |\mathbb{V}| \tag{1}$$

$$\text{Traffic}(\text{SpMM}_{\text{Pull}}) = 2 * \text{S} * S_C * (\lambda^{\mathbb{U}}_{N/L} - 1) * |\mathbb{U}| \tag{2}$$

For a general graph, $|V|$ is the total number of synchronized vertices. Thus, we have:

$$\text{Traffic}(\text{Push/Pull}) = 2 * \text{S} * S_C * (\lambda_{N/L} - 1) * |V| \tag{3}$$

Our results show that, with Hybrid-cut used as $\mathcal{P}$ in partitioning the Libimseti dataset, λ_2 equals to 1.93 and λ_{64} equals to 11.52. Hence $1-(0.93/10.52) = 91\%$ of the network traffic is reduced by partitioning the graph into 32 layers (so that in each layer just has 2 partitions) rather than 1. Figure 6a shows that the reduction on network traffic incurs a 7.78× and 7.45× speedup on average execution time when S_C is set to 256 and 1024, respectively.

5.2.2 SumV

In SpMM, the best performance is always achieved by having as many layers as possible (i.e. best L is the number of workers). This is because SpMM incurs only intra-layer communications. In contrast, for operations that require inter-layer communications, the network traffic and execution time will increase with large L. To understand this aspect, we consider a micro benchmark SumV, which computes the sum of all elements in *DColle* vector for each vertex and stores the result in the corresponding *DShare* of each vertex (i.e., $DShare_u := sum(DColle_u)$). SumV can be implemented by a single *UpdateVertex*. As we have mentioned in Section 4.2, a local combiner can be used to reduce the network traffic of SumV. However, this optimization is not used in our experiments since we intend to measure the overhead of general cases.

Figure 6b provides the execution time of SumV on 64 workers with L from 1 to 64. We see that as L increases, the execution time becomes longer, this validates our previous analysis. We also see that the slope of this curve is decreasing when L becomes larger. To explain this phenomenon, we calculate the exact amount of network traffic during the execution of one SumV. Specifically, for enabling an *UpdateVertex* operation, each master node $Node_{i,j}$ needs to gather all elements of *DColle* of v, if $v \in V_{i,j}$. Since $V_{i,j} \subseteq V_j$, the total amount of data that $Node_{i,j}$ should gather is $S_C * |V_{i,j}| - \frac{S_C}{L} * |V_{i,j}| = \frac{L-1}{L} * S_C * |V_{i,j}|$. Then, all master nodes perform the update and scatter a total amount of $(L-1)*|V|$ DShare data. As a result, the total communication cost of a SumV operation is

$$\begin{aligned} &\text{Traffic(SumV)} = \text{Traffic(UpdateVertex)} \\ &\quad = 2 * \text{S} * \frac{L-1}{L} * S_C * |V| + \text{S} * (L-1) * |V| \end{aligned} \tag{4}$$

We see that if S_C is large enough, the communication cost will be dominated by the first term, which has an upper bound and the slope of its increase becomes smaller as L becomes larger. Since the execution time is roughly decided by network traffic, we see the very similar trend in Figure 6b.

5.2.3 SumE

SumE is a similar micro benchmark to SumV, it does the same operations for all edges. Figure 6c presents the average execution time for executing a single *UpdateEdge*, which performs the equation "$DShare_{u\to v} := sum(DColle_{u\to v})$". The communication cost of SumE is almost the same as SumV, except that *DColle* of edges rather than vertices are gathered and scattered. As a result, the communication cost of a SumE operation is:

$$\begin{aligned} &\text{Traffic(SumE)} = \text{Traffic(UpdateEdge)} \\ &\quad = 2 * \text{S} * \frac{L-1}{L} * S_C * |E| + \text{S} * (L-1) * |E| \end{aligned} \tag{5}$$

As we can infer from the equation, data lines in Figure 6c share the same tendency of the lines in Figure 6b.

5.2.4 Summary

We see from the micro benchmarks that, *Update* becomes slower as L increases while *Push/Pull/Sink* becomes faster. Given a real-world algorithm which uses the basic operations in UPPS as building blocks, programmers should first obtain the replication factor of the graphs and plug it into the equations to estimate the best L that achieves lowest communication cost.

5.3 Real Applications

Besides the micro-benchmarks described above, we also implemented the GD and ALS algorithm that we explained in Section 3.6. ALS involves intra-layer communications due to *Push/Pull* and inter-layer communications due to *UpdateVertex*. GD combines the intra-layer operation *Sink* with the inter-layer operation *UpdateEdge*. The *UpdateEdge* of GD can be optimized by the local combiner while ALS cannot. ALS explores the specialized APIs for bipartite graphs while GD uses the normal ones. As a conclusion, the implementation of these two algorithms covers all common patterns of CUBE, and hence many other algorithms can be considered as some weighted combinations of GD and ALS. For example, the back-propagation algorithm for training neural networks can be implemented by combining an ALS-like round (for calculating the loss function) and a GD-like round (that updates parameters).

In the following sections, we first demonstrate the performance improvements of CUBE over the existing systems PowerGraph and PowerLyra. Then, we present a piecewise breakdown of our performance gain by calculating the network traffic reductions as in Section 5.2.

5.3.1 Implementation

Both PowerGraph and PowerLyra have provided their implementation of GD and ALS, we use *oblivious* [15] for PowerGraph and the corresponding best 2D partitioners (as listed in Table 2) for PowerLyra.

In CUBE, the implementation of GD and ALS are similar to those given in Section 3.6. However, some optimizations for further reducing network traffic are applied. For GD, we enable a local combiner for the *UpdateEdge* operation. For ALS, we merge successive *UpdateVertexU* and *UpdateVertexV* operations into one (e.g., the two *UpdateVertexV* operations at line 3 and line

Table 3: Results on execution time. Each of the cell gives data in the format of "PowerGraph / PowerLyra / CUBE" (in Second/Iteration). The number in parenthesis is the chosen L.

D	# of	Libimseti	
	workers	GD	ALS
64	8	9.78 / 9.56 / 2.04 (2)	70.8 / 70.4 / 46.7 (8)
	16	8.04 / 8.16 / 1.95 (4)	72.6 / 71.5 / 37.6 (16)
	64	6.82 / 6.89 / 2.59 (4)	87.0 / 86.8 / 28.7 (64)
128	8	14.99 / 14.94 / 3.87 (2)	261 / 258 / 193 (8)
	16	12.81 / 12.91 / 2.62 (4)	270 / 270 / 135 (16)
	64	11.64 / 11.62 / 3.33 (8)	331 / 331 / 109 (64)

D	# of	LastFM	
	workers	GD	ALS
64	8	12.0 / 8.98 / 3.45 (2)	124 / 73.5 / 70.9 (8)
	16	10.5 / 8.22 / 2.59 (2)	128 / 69.5 / 61.6 (16)
	64	10.4 / 9.86 / 2.48 (4)	158 / 111 / 57.6 (64)
128	8	19.0 / 13.8 / 4.74 (2)	465 / 263 / 270 (4)
	16	17.6 / 13.5 / 3.35 (4)	490 / 253 / 200 (16)
	64	18.6 / 17.8 / 3.47 (8)	Failed / Failed / 230 (64)

D	# of	Netflix	
	workers	GD	ALS
64	8	34.4 / 27.7 / 6.03 (1)	256 / 204 / 110 (2)
	16	26.7 / 17.3 / 3.97 (1)	186 / 107 / 60.4 (2)
	64	18.3 / 7.42 / 4.16 (1)	179 / 66.0 / 42.5 (8)
128	8	51.8 / 38.6 / 9.65 (1)	865 / 657 / 463 (1)
	16	41.9 / 23.0 / 6.59 (1)	669 / 340 / 258 (2)
	64	30.6 / 11.3 / 6.55 (2)	Failed / 239 / 118 (8)

4 of Algorithm 2 is actually implemented as one *UpdateVertexV* operation whose input function successively execute F_3 and F_1). The 2D partitioning algorithm $\mathcal{P}$ used for consisting our 3D partitioner is the listed in Table 2, and hence is the same as PowerLyra.

5.3.2 Execution Time

Table 3 shows execution time results. D is the size of the latent dimension, which gives opportunities that were not exploited in previous systems. In general, a higher D produces higher accuracy of prediction with higher both memory consumption and computational cost. We report the execution time of GD and ALS on three datasets (Libimseti, LastFM and Netflix) with three different number of workers (8, 16 and 64). For each case, we conduct the evaluation on three systems: PowerGraph [15], PowerLyra [10] and CUBE, the results are shown in the same order in the table. The number in parenthesis for CUBE indicates the chosen L for the reported execution time, which is the one with best performance. "Failed" means that the execution in this case failed due to exhausted memory.

As a summary of the results, CUBE outperforms PowerLyra by up to 4.7× and 3.1× on the GD and ALS algorithm respectively. The speedup over PowerGraph is even higher (about $7.3\times-1.5\times$). According to our analysis, the speedup on ALS is mainly caused by the reduction on network traffic, while the speedup on GD is caused by both the reduction on network traffic and the increasing of data locality. This is because that the computation part of the ALS algorithm is dominated by the DSYSV kernel, which is a CPU-bounded algorithm that has an $O(N^3)$ complexity. In contrast, the GD algorithm is mainly memory bandwidth bounded and hence is sensitive to memory locality. Next, we quantitatively discuss the network traffic of the two applications.

5.3.3 Communication Cost

As we have mentioned above, the improvement of CUBE is mainly from two aspects: *1)* reduction on network communications; and *2)* the adoption of a matrix backend. Thus, in order to further understand the performance gain, we performed a detailed analysis on the effect of network reductions, and the rests are resulted from the matrix backend.

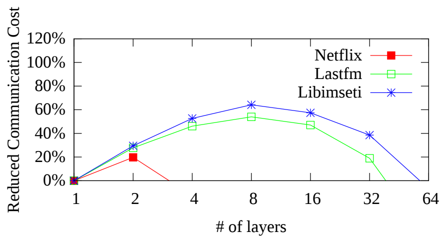


Figure 7: Reduction on GD (64 workers, $D = 128$).

GD The network traffic of a CUBE program can be calculated with the equations given in Section 5.2. But, since a local combiner is used for *UpdateEdge*, its communication cost is only $2 * 8\text{byte} * (L-1) * |E|$[3]. The network traffic for a *Sink* is half of *Push/Pull*. As a result, communication cost of each GD iteration is:

$$\begin{aligned}\text{Traffic(GD)} = (2+2+1) * 8\text{byte} * (\lambda_{N/L} - 1) * S_C * |V| \\ + 2 * 8\text{byte} * (L-1) * |E|\end{aligned} \quad (6)$$

The reduced network traffic is plotted in Figure 7, which are both the results of mathematical derivation and experimental evaluation. This is because that, as the metadata exchanged by workers account for only a negligible part of the whole communication cost, the measured results are almost identical to the number calculated by formulas. As we can see from the figure, the network traffic reduction for GD is related to replication factor, density of graph (i.e. $|E|/|V|$) and S_C. If the density large enough ($|E|/|V| \gg S_C$), the best choice is to group all nodes into one layer. It happens to be the case for Netflix dataset, which has a density of more than 200. Therefore, the best L is almost always 1 for a small D (except when D=128 and worker count is 64, i.e., the illustrated case in Figure 7). In contrast, for Libimesti, whose density is only 57, our 3D algorithm can reduce about 64% network traffic.

In order to further understand the effectiveness of our 3D partitioner, we have also performed a piecewise

[3]This result is based on Equation 5, in which $S = 8$. The first term is divided by D/L because we use a local combiner, and the second term is zero because *DShare* is *NULL*

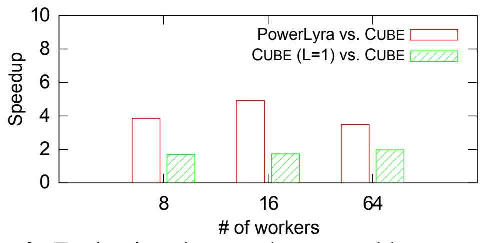


Figure 8: Evaluating the speedup caused by network reduction only for running GD on Libimesti with $D = 128$.

breakdown analysis of the speedup achieved by CUBE. This is possible, because we can estimate the performance improvements gained by 3D partitioning **only** through comparing CUBE with itself that has layer count L be fixed to 1. Figure 8 illustrates the results on Libimesti. As we can see, 3D partitioner accounts for about half of the whole speedup (up to about $2\times$). The results on Lastfm are quite similar to Libimesti but, as we can also infer from Figure 7, most of the speedup for Netflix is resulted from our matrix backend. This is why, in Table 3, the best layer count for running GD on Netflix is usually set to 1. However, if D is set to 2048, even for Netflix, the best L becomes 8 with 64 workers, which achieves a $2.5\times$ speedup compared to $L = 1$.

Moreover, since we use a matrix-based backend that is more efficient than the graph engine used in PowerGraph and PowerLyra, the **total** speedup on memory-bounded algorithms, such as GD, is still up to $4.7\times$. A similar speedup ($1.2\times - 7\times$) is reported by the single-machine graph engine GraphMat [34], which also maps a vertex program to matrix backend.

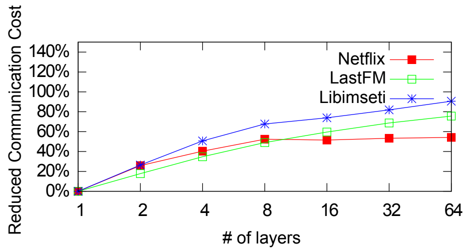


Figure 9: Reduction on ALS (64 workers, $D = 128$).

ALS As discussed in Section 5.3.1, we merged the successive *UpdateVertexU* and *UpdateVertexV* in ALS for reducing synchronizations. After the merge, each iteration of the ALS algorithm only needs to execute each of the four operations (i.e., *UpdateVertexU*, *Push*, *UpdateVertexV* and *Pull*) in bipartite mode once. Thus, based on the estimating formulas given in Section 5.2 (i.e, Equation 1, Equation 2 and Equation 4), the network traffic needed in each iteration is:

$$\begin{aligned}\text{Traffic(ALS)} =& 2*8\text{byte}*(\lambda_{N/L} - 1 + \frac{L-1}{L})*S_C*(|\mathbb{U}|+|\mathbb{V}|) \\ &+ 8\text{byte}*(L-1)*(|\mathbb{U}|+|\mathbb{V}|)\end{aligned} \quad (7)$$

According to Equation 7, we can infer that our 3D partitioner can achieve more significant network traffic reduction on a graph if it is hard to reduce replicas (i.e. λ_N is large). Figure 9 shows the relation between layer count L and the proportion of reduced network traffics when executing ALS with 64 workers and $D = 128$. For example, $\lambda_{64} = 11.52$ for Libimseti (Figure 5), thus network traffic is drastically reduced by 90.6% by partitioning the graph into 64 layers. Table 3 shows that such reduction leads to about $3\times$ speedup on the average execution time. In contrast, the replication factor for the other two datasets is relatively small and hence the speedup is also not as significant as the speedup on Libimseti.

Similar to GD, we have also performed the piecewise breakdown analysis for ALS, the results show that almost **all** ($> 90\%$) of the performance improvements are from 3D partitioning. As we have mentioned in Section 5.3.2, this is because that the computation part of the ALS algorithm is dominated by the DSYSV kernel. DSYSV is a CPU-bounded algorithm that computes the solution to a real system of linear equations, which has an $O(N^3)$ complexity and its state-of-the-art implementation has already efficiently explored its inner-operation locality. As a result, there is not much help of adopting a matrix backend.

5.4 Scalability

For many graph algorithms, the communication cost grows with the number of nodes used. Therefore, the scalability could be limited for those algorithms on small graphs. This is because that the network time may soon dominate the whole execution time, and the reduction of computation time could not offset the increase of network time.

While the potential scalability limitations exist, since our 3D partitioning algorithm reduces the network traffic, CUBE scales better than PowerGraph and PowerLyra. As we can see from Table 3, for Libimseti and LastFM, the execution time of PowerLyra actually *increases* after the number of workers reaches 16, while CUBE with lower network traffic can scale to 64 workers in most cases. Although the scalability of CUBE also becomes limited for more than 16 workers, we believe that it is mainly because that the graph size is not large enough. We expect that for those billion/trillion-edge graphs used in industry [13], our system will be able to scale to hundreds of nodes. To partially validate our hypothesis, we tested CUBE on a random generated synthetic graph, which also follows the power law and contains around one billion edges. The results show that CUBE can scale to 128 workers easily (a further $2.2\times$ speedup is achieved with 128 vs. 32 workers.). Moreover, existing techniques [3, 21] that could improve Pregel/PowerGraph's scalability can also be used to improve our system.

5.5 Memory Consumption

Table 3 shows that, L for the best performance of ALS is almost always equal to the number of workers on Libimesti and LastFM dataset. However, L affects the total memory consumption in different ways. On one side, when L increases, the size of memory for replicas of *DColle* is reduced by the partition of property vector. On the other side, the memory consumption could increase because *DShare* needs to be replicated on each layer. Specifically, since each edge has *DShare* data with type *double* in our case, the total memory needed in ALS is $(\lambda_{N/L} * S_C * |V| + L * |E|) * 8$ bytes, where $S_C = D^2 + D$. For example, Figure 10 shows the total memory consumption (the sum of the memory needed on all nodes) with different L when running ALS on Libmesti with 64 workers.

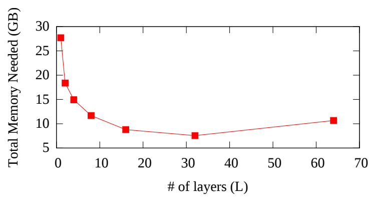


Figure 10: Total memory needed for running ALS with 64 workers and $D = 32$, $S_C = 1056$.

We see that the total memory consumption first decreases, but after a point (roughly $L = 32$) it slightly increases. The memory consumption with $L = 64$ is larger than $L = 32$, because the reduction on replicas of *DColle* data cannot offset the increase of shared *DShare* data. Therefore, $L = 64$ is the parameter for the best performance at the cost of a slightly increased memory consumption. Nevertheless, we see that the total memory consumption at $L = 1$ is *much* larger than cases when $L > 1$. Therefore, CUBE using a 3D partitioning algorithm usually consume less memory than PowerGraph and PowerLyra.

5.6 Partitioning Time

Some works [18] indicated that intelligent graph partitioning algorithms may have a dominating time and hence actually increase the total execution time. However, according to Chen et al. [10], this is only partially true for simple heuristic-based partitioning algorithms. As we can deduce from the definitions given in section 3.2, the partitioning complexity of a 3D partitioner is almost the same as the 2D partitioning algorithm. Thus it only trades a negligible growth of graph partitioning time for a notable speedup during graph computation. Moreover, for those sophisticated MLDM applications that CUBE focuses on, the ingress time typically only counts for a small partition of the overall computation time. As a result, we believe that the partitioning time of CUBE is negligible.

Specifically, the whole setup procedure of CUBE can be split into three phases namely loading, assigning and re-dispatching. *1)*. In the loading phase, each node reads an exclusive part of graph data, which is the same as most existing systems. *2)*. In the assigning phase, the assignment of each edge is calculated by the 2D partitioner $\mathcal{P}$. Since both hybrid-cut and bi-cut can calculate the assignment for each edge independently, this phase is also very fast (at least its cost is not larger than PowerLyra). Finally, *3)*. in the re-dispatching phase, each node sends its loaded data to other nodes if it is necessary (according to the data partition policy detailed in Section 3.2). Typically, the cost of sending edge data is proportional to L, while the cost of sending vertex data is negatively related to L as there are fewer replicas. If there are initial data for vertexes, the total sending cost is approximately equal to the total memory consumption. As illustrated by Figure 10, this means that setting $L > 1$ may actually reduce the cost. In contrast, if vertexes data are randomly initialized, we do have a larger cost with larger L. But, as mentioned in above, typically this cost will not exceed the communication cost of one computing iteration, and hence is acceptable.

5.7 Discussion

COST A recent study [25] shows that some distributed systems may only scale well when its single-threaded implementation has a high cost. The paper proposes a new metric COST (i.e. **C**onfiguration that **O**utperforms a **S**ingle **T**hread) to capture this type of inefficiency. If the COST of a system is c, it means that it takes c workers for this system to outperform a single-threaded implementation of the same algorithm. We also conduct COST analysis for CUBE. To do so, we built single-threaded implementations of both GD and ALS, which are just straightforward transformations of the algorithms described in Section 3.6 to BLAS operators. Based on them, we evaluate the COST of CUBE and find that it is only up to 4, which is moderate. In comparison, McSherry et al. [25] indicates that the data-parallel systems reported in recent SOSP and OSDI either have "a surprisingly *large COST, often hundreds of cores*, or simply *underperform one thread* for all of their reported configurations". Our results align with recent investigations [31, 34], which shows that matrix-centric systems usually have a much better COST than vertex-centric systems.

Faster Network The interconnect of our platform is using 1Gb Ethernet, which is a common configuration used in several recent papers [10]. Readers may wonder that whether the speedups presented is reproducible on a faster experimental setup, which is becoming more and more popular. However, according to our evaluation, when we use 10Gb network, the execution time of PowerGraph/PowerLyra is only reduced by up to 30%, such reduction is smaller than our *MPI_Alltoallv* based system. For example, when running ALS on Netflix dataset with 64 nodes and $D = 128$, the execution time of PowerLyra only reduces from 239s/iter (1Gb) to 187s/iter (10Gb). In contrast, our CUBE is accelerated from 118s/iter (1Gb) to 63.9s/iter (10Gb) (in which the time consumed by *MPI_AllToAllv* is reduced from 73.1s/iter to 17.8s/iter). Although counter-intuitive, it seems that PowerGraph/PowerLyra cannot fully utilize the network optimization and hence the speedup of CUBE over PowerGraph/PowerLyra is even bigger over a 10Gb network.

Impact of Graph Structure As mentioned in Section 5.1, structure of the input graph does have a great impact of the speedup that can be achieved by our 3D partitioning algorithm. Essentially, less skewness in graphs means that there are fewer opportunities for existing 2D partitioner (e.g., hybrid-cut, bi-cut) to explore[4]. Thus, the replica factor λ_x increases more faster with the number of sub-graphs x, which leads to a better speedup of using 3D partitioning. This is the reason that why we call a 2D partitioner "better" and use it in the evaluation if it produces fewer replicas. We want to make sure that the speedup we achieved is not based on a poor $\mathcal{P}$.

Comparison with Other Systems There are currently a variety of graph-parallel systems. Here we only concentrate on comparing with PowerLyra because its partitioning algorithm produces significant fewer replicas than the others and hence it incurs the lowest network traffic. Moreover, according to Satish et al. [31], Giraph and SociaLite [32] is slower than GraphLab, and hence will be much slower than PowerLyra. As for CombBLAS [8], due to the restriction of its programming model, both the ALS and GD algorithm can only be implemented by S_C times of SpMV in ComBLAS [31], which is extremely slow when S_C is large.

Scope of Application In general, our method is applicable to algorithms analyzing relations among divisible properties. The algorithms presented in this paper are only examples but not all we can support. As an illustration, the matrix to matrix multiplication and matrix factorization examples presented above are building blocks of many other MLDM algorithms. Thus, these problems (e.g., neural network training, mini-batched SGD, etc.) can also benefit from our method. Moreover, some algorithms, whose basic version have only indivisible properties, have advanced versions that involve divisible properties (e.g., Topic-sensitive PageRank [17], Multi-source BFS [35], etc), which obviously can also take advantage of a 3D partitioner.

[4]Many state-of-the-art 2D partitioning algorithms take advantage from the fact that most real-world graphs follow power law, hence they may not work well if the data is not that skewed.

6 Other Related Work

Several graph parallel systems [8, 10, 12, 15, 16, 26, 27, 28, 29, 30, 36, 39] have been proposed for processing the large graphs and sparse matrices. Although these systems are different from each other in terms of programming models and backend implementations, our system, CUBE, is fundamentally different from all of them by adopting a novel 3D partitioning strategy. As shown in Section 2, this 3D partitioning reduces network traffic by up to 90.6%. Besides graph partitioning, there are also many algorithms have been proposed for partitioning large matrices [4, 5]. Our 3D partitioning algorithm is inspired by the 2.5D matrix multiplication algorithm presented by Solomonik et al. [33]. However, the 2.5D algorithm is designed for multiplying two dense matrices and hence cannot be used in graph processing. Regarding the backend execution engine, GraphMat [34] provides a vertex programming frontend and maps it to a matrix backend. However, it is based on a single-machine system that cannot scale out by adding more nodes. Our system adopts the same strategy as GraphMat while extending it to a distributed environment.

7 Conclusion

We argue that the popular "task partitioning == graph partitioning" assumption is untrue for many MLDM algorithms and may result in suboptimal performance. For those MLDM algorithms, instead of a single value, a *vector* of data elements is defined as the property for each vertex/edge. We explore this feature and propose a category of *3D partitioning* algorithm that considers the hidden dimension to partition the property vector to different nodes. Based on 3D partitioning, we built CUBE, a new graph computation engine that *1)* adopts the novel 3D partitioning for reducing communication cost; *2)* provides the users with a new vertex-centric programming model UPPS; and *3)* leverages a matrix-based data structure in the backend to achieve high performance. Our evaluation results show that CUBE outperforms the existing 2D and vertex-based frameworks PowerLyra by up to 4.7× (up to 7.3× speedup over PowerGraph).

8 Acknowledgments

We want to thank Chuntao Hong and Pin Gao for their advises, all the anonymous reviewers and our shepherd for their detailed and useful reviews. This work is supported by Natural Science Foundation of China (61433008, 61373145, 61572280, 61073011, 61133004, U1435216), National Key Research & Development Program (2016YFB1000500), National Basic Research (973) Program of China (2014CB340402). Spanish Gov. European ERDF under TIN2010-21291-C02-01 and Consolider CSD2007-00050.

References

[1] S. N. A. Project. Stanford large network dataset collection. http://snap.stanford.edu/data/.

[2] Anderson, E., Bai, Z., Bischof, C., Blackford, S., Demmel, J., Dongarra, J., Du Croz, J., Greenbaum, A., Hammarling, S., McKenney, A., and Sorensen, D. *LAPACK Users' Guide*, third ed. 1999.

[3] Awara, K., Jamjoom, H., and Kanlis, P. To 4,000 compute nodes and beyond: Network-aware vertex placement in large-scale graph processing systems. SIGCOMM '13, pp. 501–502.

[4] Ballard, G., Buluc, A., Demmel, J., Grigori, L., Lipshitz, B., Schwartz, O., and Toledo, S. Communication optimal parallel multiplication of sparse random matrices. SPAA '13, pp. 222–231.

[5] Ballard, G., Demmel, J., Holtz, O., and Schwartz, O. Graph expansion and communication costs of fast matrix multiplication: Regular submission. SPAA '11, pp. 1–12.

[6] Bender, M. A., Brodal, G. S., Fagerberg, R., Jacob, R., and Vicari, E. Optimal Sparse Matrix Dense Vector Multiplication in the I/O-Model. SPAA '07, pp. 61–70.

[7] Brozovsky, L., and Petricek, V. Recommender system for online dating service. Znalosti '07.

[8] Buluç, A., and Gilbert, J. R. The combinatorial BLAS: design, implementation, and applications. *IJHPCA 25* (2011), 496–509.

[9] Celma, O. *Music Recommendation and Discovery in the Long Tail*. Springer, 2010.

[10] Chen, R., Shi, J., Chen, Y., and Chen, H. Powerlyra: Differentiated graph computation and partitioning on skewed graphs. EuroSys '15, pp. 1:1–1:15.

[11] Chen, R., Shi, J., Zang, B., and Guan, H. Bipartite-oriented distributed graph partitioning for big learning. APSys '14.

[12] 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. EuroSys '12, pp. 85–98.

[13] Ching, A., Edunov, S., Kabiljo, M., Logothetis, D., and Muthukrishnan, S. One trillion edges: Graph processing at facebook-scale. *Proc. VLDB Endow. 8*, 12 (2015), 1804–1815.

[14] Coates, A., Huval, B., Wang, T., Wu, D. J., Catanzaro, B. C., and Ng, A. Y. Deep learning with COTS HPC systems. ICML '13, pp. 1337–1345.

[15] Gonzalez, J. E., Low, Y., Gu, H., Bickson, D., and Guestrin, C. Powergraph: Distributed graph-parallel computation on natural graphs. OSDI '12, pp. 17–30.

[16] Gonzalez, J. E., Xin, R. S., Dave, A., Crankshaw, D., Franklin, M. J., and Stoica, I. Graphx: Graph processing in a distributed dataflow framework. OSDI '14, pp. 599–613.

[17] Haveliwala, T. H. Topic-sensitive pagerank. In *Proceedings of the 11th International Conference on World Wide Web* (New York, NY, USA, 2002), WWW '02, ACM, pp. 517–526.

[18] Hoque, I., and Gupta, I. Lfgraph: Simple and fast distributed graph analytics. In *Proceedings of the First ACM SIGOPS Conference on Timely Results in Operating Systems* (New York, NY, USA, 2013), TRIOS '13, ACM, pp. 9:1–9:17.

[19] Huang, C.-C., Chen, Q., Wang, Z., Power, R., Ortiz, J., Li, J., and Xiao, Z. Spartan: A distributed array framework with smart tiling. USENIX ATC '15, pp. 1–15.

[20] Jannach, D., Zanker, M., Felfernig, A., and Friedrich, G. *Recommender systems: an introduction*. Cambridge University Press, 2010.

[21] Khayyat, Z., Awara, K., Alonazi, A., Jamjoom, H., Williams, D., and Kalnis, P. Mizan: A system for dynamic load balancing in large-scale graph processing. EuroSys '13, pp. 169–182.

[22] Low, Y., Bickson, D., Gonzalez, J., Guestrin, C., Kyrola, A., and Hellerstein, J. M. Distributed graphlab: A framework for machine learning and data mining in the cloud. *Proc. VLDB Endow. 5* (2012), 716–727.

[23] 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. SIGMOD '10, pp. 135–146.

[24] McSherry, F. Spectral partitioning of random graphs. FOCS '01, pp. 529–.

[25] McSherry, F., Isard, M., and Murray, D. G. Scalability! But at What Cost? In *Proceedings of the 15th USENIX Conference on Hot Topics in Operating Systems* (Berkeley, CA, USA, 2015), HOTOS'15, USENIX Association, pp. 14–14.

[26] Murray, D. G., McSherry, F., Isaacs, R., Isard, M., Barham, P., and Abadi, M. Naiad: A timely dataflow system. SOSP '13, pp. 439–455.

[27] Prabhakaran, V., Wu, M., Weng, X., McSherry, F., Zhou, L., and Haridasan, M. Managing large graphs on multi-cores with graph awareness. USENIX ATC'12, pp. 4–4.

[28] Pundir, M., Leslie, L. M., Gupta, I., and Campbell, R. H. Zorro: Zero-cost reactive failure recovery in distributed graph processing. In *Proceedings of the Sixth ACM Symposium on Cloud Computing*, SoCC '15, pp. 195–208.

[29] Roy, A., Bindschaedler, L., Malicevic, J., and Zwaenepoel, W. Chaos: Scale-out graph processing from secondary storage. SOSP '15, pp. 410–424.

[30] Roy, A., Mihailovic, I., and Zwaenepoel, W. X-stream: Edge-centric graph processing using streaming partitions. SOSP '13, pp. 472–488.

[31] Satish, N., Sundaram, N., Patwary, M. M. A., Seo, J., Park, J., Hassaan, M. A., Sengupta, S., Yin, Z., and Dubey, P. Navigating the maze of graph analytics frameworks using massive graph datasets. SIGMOD '14, pp. 979–990.

[32] Seo, J., Park, J., Shin, J., and Lam, M. S. Distributed socialite: A datalog-based language for large-scale graph analysis. *Proc. VLDB Endow. 6*, 14 (2013), 1906–1917.

[33] Solomonik, E., and Demmel, J. Communication-optimal Parallel 2.5D Matrix Multiplication and LU Factorization Algorithms. Euro-Par '11, pp. 90–109.

[34] Sundaram, N., Satish, N., Patwary, M. M. A., Dulloor, S. R., Anderson, M. J., Vadlamudi, S. G., Das, D., and Dubey, P. GraphMat: High performance graph analytics made productive. *Proc. VLDB Endow. 8*, 11 (2015), 1214–1225.

[35] Then, M., Kaufmann, M., Chirigati, F., Hoang-Vu, T.-A., Pham, K., Kemper, A., Neumann, T., and Vo, H. T. The more the merrier: Efficient multi-source graph traversal. *Proc. VLDB Endow. 8*, 4 (Dec. 2014), 449–460.

[36] Wu, M., Yang, F., Xue, J., Xiao, W., Miao, Y., Wei, L., Lin, H., Dai, Y., and Zhou, L. Gram: Scaling graph computation to the trillions. SoCC '15, pp. 408–421.

[37] Zhang, M., Wu, Y., Chen, K., Ma, T., and Zheng, W. Measuring and optimizing distributed array programs. *Proc. VLDB Endow. 9*, 12 (Aug. 2016), 912–923.

[38] Zhou, Y., Wilkinson, D., Schreiber, R., and Pan, R. Large-scale parallel collaborative filtering for the netflix prize. AAIM '08, pp. 337–348.

[39] Zhu, X., Chen, W., Zheng, W., and Ma, X. Gemini: A computation-centric distributed graph processing system. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)* (Savannah, GA, Nov. 2016), USENIX Association.

Gemini: A Computation-Centric Distributed Graph Processing System

Xiaowei Zhu[1], Wenguang Chen[1,2,*], Weimin Zheng[1], and Xiaosong Ma[3]

[1]Department of Computer Science and Technology (TNLIST), Tsinghua University
[2]Technology Innovation Center at Yinzhou,
Yangtze Delta Region Institute of Tsinghua University, Zhejiang
[3]Qatar Computing Research Institute, Hamad Bin Khalifa University

Abstract

Traditionally distributed graph processing systems have largely focused on scalability through the optimizations of inter-node communication and load balance. However, they often deliver unsatisfactory overall processing efficiency compared with shared-memory graph computing frameworks. We analyze the behavior of several graph-parallel systems and find that the added overhead for achieving scalability becomes a major limiting factor for efficiency, especially with modern multi-core processors and high-speed interconnection networks.

Based on our observations, we present Gemini, a distributed graph processing system that applies multiple optimizations targeting computation performance to *build scalability on top of efficiency*. Gemini adopts (1) a sparse-dense signal-slot abstraction to extend the hybrid push-pull computation model from shared-memory to distributed scenarios, (2) a chunk-based partitioning scheme enabling low-overhead scaling out designs and locality-preserving vertex accesses, (3) a dual representation scheme to compress accesses to vertex indices, (4) NUMA-aware sub-partitioning for efficient intra-node memory accesses, plus (5) locality-aware chunking and fine-grained work-stealing for improving both inter-node and intra-node load balance, respectively. Our evaluation on an 8-node high-performance cluster (using five widely used graph applications and five real-world graphs) shows that Gemini significantly outperforms all well-known existing distributed graph processing systems, delivering up to 39.8× (from 8.91×) improvement over the fastest among them.

1 Introduction

Graph processing is gaining increasing attentions in both academic and industrial communities. With the magnitude of graph data growing rapidly, many specialized distributed systems [3, 12, 16, 17, 30, 32, 41, 45] have been proposed to process large-scale graphs.

While these systems are able to take advantage of multiple machines to achieve scalability, their performance is often unsatisfactory compared with state-of-the-art shared-memory counterparts [36, 47, 49, 57]. Further, a recent study [33] shows that an optimized single-thread implementation is able to outperform many distributed systems using many more cores. Our hands-on experiments and performance analysis reveal several types of design and implementation deficiencies that lead to loss of performance (details in Section 2).

Based on the performance measurement and code examinations, we come to recognize that traditional distributed graph-parallel systems do not fit in today's powerful multi-core cluster nodes and fast-speed networks. To achieve better overall performance, one needs to focus on the performance of both computation and communication components, compressing the computation time aggressively while hiding the communication cost, rather than focusing primarily on minimizing communication volume, as seen in multiple existing systems' design.

To bridge the gap between efficient shared-memory and scalable distributed systems, we present Gemini, a distributed graph processing system that builds *scalability* on top of *efficiency*. More specifically, the main contributions of this work are summarized as follows:

- We perform detailed analysis of several existing shared-memory and distributed graph-parallel systems and identify multiple design pitfalls.
- We recognize that efficient and scalable distributed graph processing involves intricate interplay between the properties of the application, the underlying system, and the input graph. In response, we explore adaptive runtime choices, such as a density-aware dual-mode processing scheme and multiple locality-aware data distribution and load balancing mechanisms. The result is a system that can deliver competitive performance on a range of system

*Corresponding author (cwg@tsinghua.edu.cn).

scales, from multi-core to multi-node platforms.

- We identify a simple yet surprisingly effective chunk-based graph partitioning scheme, which facilitates exploitation of natural locality in input graphs and enables seamless hierarchical refinement. We present multiple optimizations enabled by this new partitioning approach.
- We evaluate our Gemini prototype with extensive experiments and compared it with five state-of-the-art systems. Experiments with five applications on five real-world graphs show that Gemini significantly outperforms existing distributed implementations, delivering up to 39.8× (from 8.91×) improvement over the fastest among them. We collect detailed measurement for performance analysis and validating internal design choices.

2 Motivation

While state-of-the-art shared-memory graph processing systems are able to process graphs quite efficiently, the lack of scalability makes them fail to handle graphs that do not fit in the memory of a single machine. On the other hand, while existing distributed solutions can scale graph processing to larger magnitudes than their shared-memory counterparts, their performance and cost efficiencies are often unsatisfactory [33, 59, 60].

To study the performance loss, we profiled several representative graph-parallel systems, including Ligra [47], Galois [36], PowerGraph [16], PowerLyra [12], as well as the optimized single-thread implementation proposed in the COST paper [33] for reference. We set up experiments on an 8-node high-performance cluster interconnected with Infiniband EDR network (with up to 100Gbps bandwidth), each node containing two Intel Xeon E5-2670 v3 CPUs (12 cores and 30MB L3 cache per CPU) and 128 GB DRAM. We ran 20 iterations of PageRank [38] on the *twitter-2010* [28] graph, a test case commonly used for evaluating graph-parallel systems.

Cores	1	24 × 1		24 × 8	
System	OST	Ligra	Galois	PowerG.	PowerL.
Runtime (s)	99.9	21.9	19.3	40.3	26.9
Instructions	525G	496G	482G	7.15T	6.06T
Mem. Ref.	15.8G	32.3G	23.4G	95.8G	87.2G
Comm. (GB)	-	-	-	115	38.1
IPC	1.71	0.408	0.414	0.500	0.655
LLC Miss	8.77%	43.9%	49.7%	71.0%	54.9%
CPU Util.	100%	91.7%	96.8%	65.5%	68.4%

Table 1: Sample performance analysis of existing systems (20 iterations of PageRank on *twitter-2010*). OST refers to the optimized single-thread implementation.

Table 1 gives detailed performance metrics for the five targeted systems. Overall, systems lose efficiency as we move from single-thread to shared memory, then to distributed implementations. Though this is to be expected with communication/synchronization overhead, load balance issues, and in general higher software complexities, the large span in almost all measurement categories across alternative systems indicates a large room for improvement.

As seen from the profiling results, the network is far from saturated (*e.g.*, lower than 3Gbps average aggregate bandwidth usage with PowerGraph). Computation, rather than communication, appears to be the actual bottleneck of evaluated distributed systems, which echoes recent findings on distributed data analytics frameworks [37]. Compared with their shared-memory counterparts, they have significantly more instructions and memory references, poorer access localities, and lower multi-core utilization. We further dig into the code and find that such inefficiency comes from multiple sources, such as (1) the use of hash maps to convert vertex IDs between global and local states, (2) the maintenance of vertex replicas, (3) the communication-bound apply phase in the GAS abstraction [16], and (4) the lack of dynamic scheduling. They either enlarge the working set, producing more instructions and memory references, or prevent the full utilization of multi-core CPUs.

We argue that many of the above side-effects could be avoided when designing distributed graph-parallel systems, by building *scalability* on top of *efficiency*, instead of focusing on the former in the first place. The subsequent distributed system design should pay close attention to the computation overhead of cross-node operation over today's high-speed interconnect, as well as the local computation efficiency on partitioned graphs.

To this end, we adapt Ligra's hybrid push-pull computation model to a distributed form, which facilitates efficient vertex-centric data update and message passing. A chunk-based partitioning scheme is adopted, allowing low-overhead graph distribution as well as recursive application at multiple system levels. We further deploy multiple optimizations to aggressively compress the computation time. Finally, we design a co-scheduling mechanism to overlap computation and inter-node communication tasks.

3 Gemini Graph Processing Abstraction

Viewing modern clusters as small or moderate number of nodes interconnected with fast networks similar to a shared-memory multi-core machine, Gemini adopts a graph processing abstraction that enables a smooth extension of state-of-the-art single-node graph computation models to cluster environments.

Before getting to details, let us first give the targeted graph processing context. Like assumed in many graph-parallel systems or frameworks, single-node [47, 59, 60] or distributed [11, 18, 23, 50] alike, a graph processing problem updates information stored in vertices, while edges are viewed as immutable objects. Also, like common systems, Gemini processes both directed and undirected graphs, though the latter could be converted to directed ones by replacing each undirected edge with a pair of directed edges. The rest of our discussion therefore assumes all edges are directed.

For a common graph processing application, the processing is done by propagating vertex updates along the edges, until the graph state converges or a given number of iterations are completed. Vertices with ongoing updates are called *active vertices*, whose outgoing edges collectively form the *active edge set* for processing.

3.1 Dual Update Propagation Model

At a given time during graph processing, the active edge set may be *dense* or *sparse*, typically determined by its size (total number of outgoing edges from active vertices) relative to $|E|$, the total number of edges. For example, the active edge set of the *CC* (connected components) application is dense in the first few iterations, and gets increasingly sparse as more vertices receive their final labels. *SSSP* (single-source shortest paths), on the other hand, starts from a very sparse edge set, getting denser as more vertices become activated by their in-neighbors, and sparse again when the algorithm approaches the final convergence.

State-of-the-art shared-memory graph processing systems [4, 36, 47, 57] have recognized that different active edge set densities call for different update propagation models. More specifically, sparse active edge sets prefer the *push* model (where updates are passed to neighboring vertices along outgoing edges), as the system only traverses outgoing edges of *active* vertices where new updates are made. In contrast, dense active edge sets benefit more from the *pull* model (where each vertex's update is done by collecting states of neighboring vertices along incoming edges), as this significantly reduces the contention in updating vertex states via locks or atomic operations.

While Ligra [47] proposed the adaptive switch between these two modes according to the density of an active edge set in a shared-memory machine (with the default threshold $|E|/20$, which Gemini follows), here we explore the feasibility of extending such design to distributed systems. The major difference is that a graph will be partitioned and distributed across different nodes, where information and updates are shared using explicit message passing. To this end, Gemini uses the master-mirror notion as in PowerGraph [16]: each vertex is assigned to (*owned by*) one partition, where it is a *master* vertex, as the primary copy maintaining vertex state data. The same vertex may also have replicas, called *mirrors*, on each node/partition that owns at least one of its neighbors. A pair of directed edges will be created between each master-mirror pair, though only one of them will be used in either propagation mode. Note that unlike in PowerGraph, mirrors in Gemini act like placeholders only for update propagation and do not hold actual data.

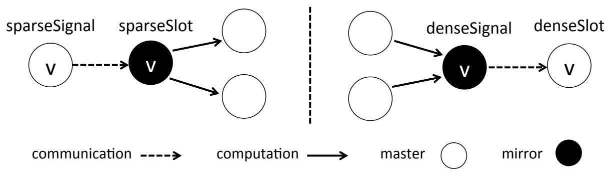


Figure 1: The sparse-dense signal-slot model

With replicated vertices, Gemini adopts a sparse-dense dual engine design, using a signal-slot abstraction to decouple the propagation of vertex states (communication) from the processing of edges (computation). Borrowed but slightly different from in the Qt software framework [1], *signals* and *slots* denote user-defined vertex-centric functions describing message sending and receiving behaviors, respectively. Computation and communication are handled differentially in the two modes, as illustrated in Figure 1. In the sparse (*push*) mode, each master first sends messages containing latest vertex states to its mirrors via `sparseSignal`, who in turn update their neighbors through outgoing edges via `sparseSlot`. In the dense (*pull*) mode, each mirror first performs local computation based on states of neighboring vertices through incoming edges, then sends an update message containing the result to its master via `denseSignal`, who subsequently updates its own state appropriately via `denseSlot`.

An interesting feature of the proposed abstraction is that *message combining* [32] is automatically enabled. Only one message per active master-mirror pair of each vertex is needed, lowering the number of messages from $O(|E|)$ to $O(|V|)$. This also allows computation to be performed locally to aggregate outgoing updates without adopting an additional "combining pass", which is necessary in many Pregel-like systems [3, 32, 44].

3.2 Gemini API

Gemini adopts an API design (Figure 2) similar to those presented by shared-memory systems [47, 57]. Data and computation distribution details are hidden from users. A graph is described in its entirety with a type `E` for edge data, and several user-defined vertex arrays. A compact `VertexSet` data structure (internally implemented with

```
class Graph<E> {
  VertexID vertices;
  EdgeID edges;
  VertexID [] outDegree;
  VertexID [] inDegree;
  def allocVertexArray<V>() -> V [];
  def allocVertexSet() -> VertexSet;
  def processVertices<A> (
    work: (VertexID) -> A,
    active: VertexSet,
    reduce: (A, A) -> A,
  ) -> A;
  def processEdges<A, M> (
    sparseSignal: (VertexID) -> void,
    sparseSlot: (VertexID, M, OutEdgeIterator<E>) -> A,
    denseSignal: (VertexID, InEdgeIterator<E>) -> void,
    denseSlot: (VertexID, M) -> A,
    reduce: (A, A) -> A,
    active: VertexSet
  ) -> A;
  def emit<M> (recipient: VertexID, message: M) -> void;
};
```

Figure 2: Core Gemini API

bitmaps) is provided for efficient representation of a vertex subset, *e.g.*, the active vertex set.

The only major Gemini APIs for users to provide custom codes are those specifying computation tasks, namely the `processVertices` and `processEdges` collections. For the rest of this section we illustrate the semantics of these user-defined functions using *CC* as an example.

```
Graph<empty> g (...); // load a graph from the file system
VertexSet activeCurr = g.allocVertexSet();
VertexSet activeNext = g.allocVertexSet();
activeCurr.fill(); // add all vertices to the set
VertexID [] label = g.allocVertexArray <VertexID> ();
def add (VertexID a, VertexID b) : VertexID {
  return a + b;
}
def initialize (VertexID v) : VertexID {
  label[v] = v;
  return 1;
}
VertexID activated = g.processVertices <VertexID> (
  initialize,
  activeCurr
);
```

Figure 3: Definitions and initialization for *CC*

As illustrated in Figure 3, we first create active vertex sets for the current/next iteration, define the `label` vertex array, and initialize the latter with own vertex IDs through a `processVertices` call.

A classic iterative label propagation method is then used to compute the connected components. Figure 4 gives the per-iteration update logic defined in two

```
def CCSparseSignal (VertexID v) {
  g.emit(v, label[v]);
}
def CCSparseSlot (VertexID v, VertexID msg, OEI iter) : VertexID
{
  VertexID activated = 0;
  while (iter.hasNext()) {
    VertexID dst = iter.next().neighbour;
    if (msg < label[dst] && atomicWriteMin(label[dst], msg)) {
      activeNext.add(dst); // add 'dst' to the next frontier
      activated += 1;
    }
  }
  return activated;
}
def CCDenseSignal (VertexID v, IEI iter) : void {
  VertexID msg = v;
  while (iter.hasNext()) {
    VertexID src = iter.next().neighbour;
    msg = msg < label[src] ? msg : label[src];
  }
  if (msg < v) g.emit(v, msg);
}
def CCDenseSlot (VertexID v, VertexID msg) : VertexID {
  if (msg < label[v] && atomicWriteMin(label[v], msg)) {
    activeNext.add(v); // add 'v' to the next frontier
    return 1;
  }
  else return 0;
}
while (activated>0) {
  activeNext.clear(); // make an empty vertex set
  activated = g.processEdges <VertexID, VertexID> (
    CCSparseSignal,
    CCSparseSlot,
    CCDenseSignal,
    CCDenseSlot,
    activeCurr,
    add
  );
  swap(activeCurr, activeNext);
}
```

Figure 4: Iterative label propagation for *CC*. `OEI` and `IEI` are edge iterators for outgoing edges and incoming edges respectively. `atomicWriteMin`(a, b) atomically assigns b to a if $b < a$. `swap`(a, b) exchanges a and b.

`signal-slot` pairs. In the sparse mode, every active vertex first broadcasts its current label from the master to its mirrors, including the master itself (`CCSparseSignal`). When a mirror receives the label, it iterates over its *local* outgoing edges and updates the vertex states of its neighbors (`CCSparseSlot`). In the dense mode, each vertex (both masters and mirrors, active or inactive) first iterates over its *local* incoming edges and sends the smallest label from the neighborhood to its master (`CCDenseSignal`). The master updates its own vertex state, upon receiving a label smaller than its current one (`CCDenseSlot`). The number of overall vertex activations in the current iteration is collected, via aggregating the `slot` function return values using `add`, to determine whether convergence has been reached.

Note that in Gemini, graph partitioning stops at the socket level. Cores on the same socket do not communicate via message passing, but directly perform updates on shared graph data. Therefore, as shown in the example, atomic operations are used in *slots* to ensure that vertex states are updated properly.

Not all user-defined functions are mandatory, *e.g.*, `reduce` is not necessary where there is no aggregation. Also, Gemini's dual mode processing is optional: users may choose to supply only the sparse or dense mode algorithm implementation, especially if it is known that staying at one mode delivers adequate performance (such as dense mode for PageRank), or when the memory is only able to hold edges in one direction.

4 Distributed Graph Representation

While Gemini's computation model presents users with a unified logical view of the entire graph, when deployed on a high-performance cluster, the actual graph has to be partitioned and distributed internally to exploit parallelism. A number of partitioning methods have been proposed, including both *vertex-centric* [30, 32, 48] and *edge-centric* (*aka. vertex-cut*) [8, 12, 16, 24, 39] solutions. Vertex-centric solutions enable a centralized computation model, where vertices are evenly assigned to partitions, along with their associated data, such as vertex states and adjacent edges. Edge-centric solutions, on the other hand, evenly assign edges to partitions and replicate vertices accordingly.

However, as profiling results in Section 2 demonstrated, prior studies focused on partitioning for load balance and communication optimizations, without paying enough attention on the resulted system complexity and the implication of partitioning design choices on the efficiency of computation.

To achieve scalability while maintaining efficiency, we propose a lightweight, chunk-based, multi-level partitioning scheme. We present several design choices regarding graph partitioning and internal representation that aim at improving the *computation performance* in distributed graph processing.

4.1 Chunk-Based Partitioning

The inspiration of Gemini's chunk-based partitioning comes from the fact that many large-scale real-world graphs often possess natural locality, with crawling being the common way to collect them in the first place. Adjacent vertices likely to be stored close to each other. Partitioning the vertex set into contiguous chunks could effectively preserve such locality. For example, in typical web graphs the lexicographical URL ordering guarantees that most of the edges connect two vertices close to each other (in vertex ID) [7]; in the Facebook friendship network, most of the links are close in geo-locations [52]. When locality happens to be lost in the input, there also exist effective and affordable methods to "recover" locality from the topological structure [2, 5], bringing the benefit of chunk-based partitioning for potentially repeated graph processing at a one-time pre-processing cost.

On a p-node cluster, a given global graph $G = (V, E)$ will be partitioned into p subgraphs $G_i = (V_i', E_i)$ (i from 0 to $p-1$), where V_i' and E_i are the vertex subset and the edge subset on the ith partition, respectively. To differentiate master vertices from others, we denote V_i to be the owned vertex subset on the ith partition.

Gemini partitions G using a simple chunk-based scheme, dividing V into p contiguous vertex chunks $(V_0, V_1, ..., V_{p-1})$, whose sizes are determined by additional optimizations discussed later in this section. Further, we use E_i^S and E_i^D to represent the outgoing and incoming edge set of partition i, used in the sparse and dense mode respectively. Each chunk (V_i) is assigned to one cluster node, which owns all vertices in this chunk. Edges are then assigned by the following rules:

$$E_i^S = \{(src, dst, value) \in E | dst \in V_i\}$$

$$E_i^D = \{(src, dst, value) \in E | src \in V_i\}$$

where *src*, *dst*, and *value* represent an edge's source vertex, destination vertex, and edge value, respectively. In other words, for the ith partition, the outgoing edge set E_i^S contains edges *destined to* its owned vertices V_i, while the incoming edge set E_i^D contains edges *sourced from* V_i.

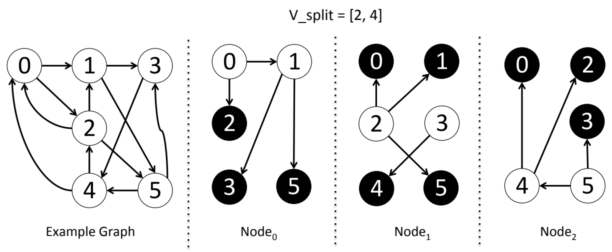


Figure 5: An example of chunk-based partitioning (dense mode), where the ID-ordered vertex array is split into three chunks $\{0,1\}, \{2,3\}, \{4,5\}$. Again black and white vertices denote mirrors and masters respectively.

Figure 5 gives an example of chunk-based partitioning, showing the vertex set on three nodes, with their corresponding dense mode edge sets. Here mirrors are created for all remote vertices that local masters have *out* edges to. These mirrors will "pull" local neighboring states to update their remote masters. The sparse mode edge sets are similar and omitted due to space limit.

With chunk-based partitioning, Gemini achieves scalability with little overhead. The contiguous partitioning enables effortless vertex affiliation queries, by simply checking partition boundaries. The contiguous feature within each chunk also simplifies vertex data representation: only the owned parts of vertex arrays are actually touched and allocated in contiguous memory pages on each node. Therefore, the memory footprint is well controlled and no vertex ID conversions are needed to compress the space consumption of vertex states.

As accesses to neighboring states generate random accesses in both push and pull modes, vertex access locality is often found to be performance-critical. Chunk-based partitioning naturally preserves the vertex access locality, which tends to be lost when random-based distribution is used. Moreover, random accesses to vertex states all falls into the owned chunk V_i rather than V or V_i'. Gemini can then benefit from chunk-based partitioning when the system scales out, where random accesses could be handled more efficiently as the chunk size decreases.

Such lightweight chunk-based partitioning does sacrifice balanced edge distribution or minimized cut edge set, but compensates for such deficiency by (1) low-overhead scaling out designs, (2) preserved memory access localities, and (3) additional load balancing and task scheduling optimizations to be presented later in the paper.

4.2 Dual-Mode Edge Representation

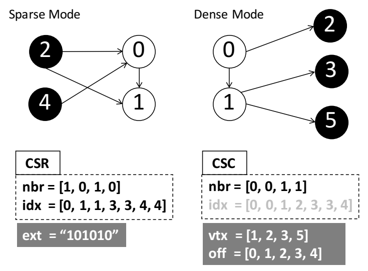


Figure 6: Sample representation of sparse/dense mode edges using CSR/CSC, with Gemini enhancement highlighted

Gemini organizes outgoing edges in the *Compressed Sparse Row* (*CSR*) and incoming ones in the *Compressed Sparse Column* (*CSC*) format. Both are compact sparse matrix data structures commonly used in graph systems, facilitating efficient vertex-centric sequential edge access. Figure 6 illustrates the graph partition on cluster node 0 from the sample graph in Figure 5 and its CSR/CSC representation to record edges adjacent to owned vertices (0 and 1). The index array `idx` records each vertex's edge distribution: for vertex i, `idx[i]` and `idx[i+1]` indicate the beginning and ending offsets of its outgoing/incoming edges to this particular partition. The array `nbr` records the neighbors of these edges (sources for incoming edges or destinations for outgoing ones).

Yet, from our experiments and performance analysis, we find that the basic CSR/CSC format is insufficient. More specifically, the index array `idx` can become a scaling bottleneck, as its size remains at $O(|V|)$ while the size of edge storage is reduced proportionally at $O(|E|/p)$ as p grows. For example, in Figure 6, the partition has only 4 dense mode edges, but has to traverse the 7-element $(|V|+1)$ `idx` array, making the processing of adjacent vertices (rather than edges) the bottleneck in dense mode computation. A conventional solution to this is to compress the vertex ID space. This comes at the cost of converting IDs between global and local states, which adds other non-negligible overhead to the system.

To resolve the bottleneck in a lightweight fashion, we use two schemes for enhancing the index array in the two modes, as described below and illustrated in Figure 6:

- *Bitmap Assisted Compressed Sparse Row*: for sparse mode edges, we add an existence bitmap `ext`, which marks whether each vertex has outgoing edges in this partition. For example, only vertex 0, 2, and 4 are present, indicated by the bitmap 101010.
- *Doubly Compressed Sparse Column*: for dense mode edges, we use a doubly-compression scheme [9] to store only vertices with incoming edges (`vtx`) and their corresponding edge offsets (`off`, where (`off[i+1]-off[i]`) indicates the number of local incoming edges vertex `vtx[i]` has). For example, only vertex 1, 2, 3, and 5 has local incoming edges.

Both schemes reduce memory accesses required in edge processing. In the dense mode, where all the vertices in a local partition has to be processed, the compressed indices enable Gemini to only access $O(|V_i'|)$ vertex indices reduced from $O(|V|)$. In the sparse mode, the bitmap eliminates the lookups into `idx` of vertices that do not have outgoing edges in the local partition, which occurs frequently when the graph is partitioned.

4.3 Locality-Aware Chunking

We now discuss how Gemini actually decides where to make the $p-1$ cuts when creating p contiguous vertex chunks, using a locality-aware criterion.

Traditionally, graph partitioning pursues *even distribution* of either the vertices (in vertex-centric scenarios) or the edges (in edge-centric scenarios) to enhance load balance.

While Gemini's chunk-based partitioning is vertex-centric, we find that balanced vertex distribution exhibits poor load balance due to the power-law distribution [15] of vertex degrees exhibited in most real-world graphs, often considered a disadvantage of vertex-centric solutions compared with their edge-centric counterparts [16].

However, with chunk-based partitioning, even balanced edge chunking, with $|E|/p$ edges per partition uniformly brings significant load imbalance. Our closer examination finds that vertex access locality (one of the performance focal points of Gemini's chunk-based partitioning) differs significantly across partitions despite balanced edge counts, incurring large variation in $|V_i|$, the size of vertex chunks.

While dynamic load balancing techniques [26, 41] such as workload re-distribution might help, they incur heavy costs and extra complexities that go against Gemini's lightweight design. Instead, Gemini employs a locality-aware enhancement, adopting a hybrid metric that considers both owned vertices and dense mode edges in setting the balancing criteria. More specifically, the vertex array V is split in a manner so that each partition has a balanced value of $\alpha \cdot |V_i| + |E_i^D|$. Here α is a configurable parameter, set empirically to $8 \cdot (p-1)$ as Gemini's default configuration in our experiments, which might be adjusted according to hardware configurations or application/input properties.

The intuition behind such hybrid metric is that one needs to take into account the computation complexity from both the vertex and the edge side. Here the size of the partition, in terms of $|V_i|$ and $|E_i^D|$, not only affects the amount of work ($|E_i^D|$), but also the memory access locality ($|V_i|$). To analyze the joint implication of specific system, algorithm, and input features on load balancing and enable adaptive chunk partitioning (*e.g.*, automatic configuration of α) is among our ongoing investigations.

4.4 NUMA-Aware Sub-Partitioning

An interesting situation with today's high-performance cluster is that the scale of intra-node parallelisms could easily match or exceed that of inter-node levels. For instance, our testbed has 8 nodes, each with 24 cores. Effectively exploiting both intra- and inter-node hardware parallelism is crucial to the overall performance of distributed graph processing.

Most modern servers are built on the NUMA (Non-Uniform Memory Access) architecture, where memory is physically distributed on multiple sockets, each typically containing a multi-core processor with local memory. Sockets are connected through high-speed interconnects into a global cache-coherent shared-memory system. Access to local memory is faster than to remote memory (attached to other sockets), both in terms of lower latencies and higher bandwidths [14], making it appealing to minimize *inter-socket* accesses.

Gemini's chunk-based graph partitioning demonstrates another advantage here, by allowing the system to recursively apply sub-partitioning in a consistent manner, potentially with different optimizations applicable at each particular level. Within a node, Gemini applies NUMA-aware sub-partitioning across multiple sockets: for each node containing s sockets, the vertex chunk V_i is further cut into *s sub-chunks*, one for each socket. Edges are assigned to corresponding sockets, using the same rules as in inter-node partitioning (Section 4.1).

NUMA-aware sub-partitioning boosts the performance on NUMA machines significantly. It retains the natural locality present in input vertex arrays, as well as lightweight partitioning and bookkeeping. With smaller yet densely processed sub-chunks, both sequential accesses to edges and random accesses to vertices are likely to fall into the local memory, facilitating faster memory access and higher LLC (last level cache) utilization simultaneously.

5 Task Scheduling

Like most recent distributed graph processing systems [3, 11, 12, 16, 17, 23, 26, 32, 41, 43], Gemini follows the Bulk Synchronous Parallel (BSP) model [53]. In each iteration of edge processing, Gemini co-schedules computation and communication tasks in a cyclic ring order to effectively overlap inter-node communication with computation. Within a node, Gemini employs a fine-grained work-stealing scheduler with shared pre-computed chunk counters to enable dynamic load balancing at a fine granularity. Below we discuss these two techniques in more detail.

5.1 Co-Scheduling of Computation and Communication Tasks

Inspired by the well-optimized implementation of collective operations in HPC communication libraries, such as `AllGather` in MPI, Gemini organizes cluster nodes in a ring, with which message sending and receiving operations are coordinated in a balanced cyclic manner, to reduce network congestion and maximize aggregate message passing throughput. Such orchestrated communication tasks are further carefully overlapped with computation tasks, to hide network communication costs.

On a cluster node with c cores, Gemini maintains an OpenMP pool of c threads for parallel vertex-centric edge processing, performing the `signal` and `slot` tasks. Each thread is bound to specific sockets to work with NUMA-aware sub-partitioning. In addition, two helper threads per node are created for inter-node message sending/receiving operations via MPI.

Here again thanks to Gemini's chunk-based partitioning and CSR/CSC organization of edges, it can naturally batch messages destined to the same partition in both sparse and dense modes for high-performance communication. Moreover, the batched messages enable us to schedule the tasks in a simple partition-oriented fashion. Figure 7 illustrates the co-scheduled ordering of the four types of tasks, using the dense mode in the first partition (*node*$_0$) of the previous figure as an example.

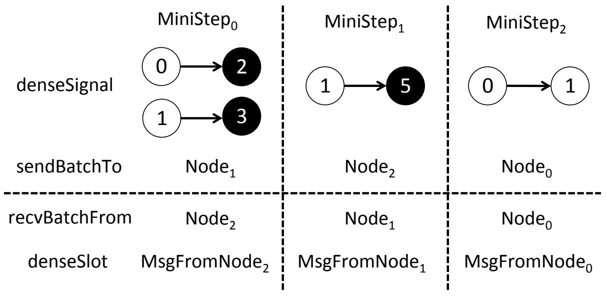


Figure 7: Example of co-scheduled computation and communication tasks on *node*$_0$

The iteration is divided into p mini-steps, during each of which *node*$_i$ communicate with one peer node, starting from *node*$_{i+1}$ back to itself. In the particular example shown in Figure 7, there are three such stages, where *node*$_0$ communicates with *node*$_1$, *node*$_2$, and *node*$_0$ respectively. In each mini-step, the node goes through local `denseSignal` processing, message send/receive, and final local `denseSlot` processing. For example, here in the first mini-step, local mirrors of vertices 2 and 3 (owned by *node*$_1$) pull updates from vertices 0 and 1 (owned by self), creating a batched message ready for *node*$_1$, after whose transmission *node*$_0$ expects a similar batched message from *node*$_2$ and processes that in `denseSlot`. In the next mini-step, similar update is pulled by all local mirrors owned by *node*$_2$ (only vertex 5 in this figure), followed by communication and local processing. The process goes on until *node*$_0$ finally "communicates" with itself, where it simply pulls from locally owned neighbors (vertex 1 from 0). As separate threads are created to execute the CPU-light message passing tasks, computation is effectively overlapped with communication.

5.2 Fine-Grained Work-Stealing

While inter-node load balance is largely ensured through the locality-aware chunk-based partitioning in Gemini, the hybrid vertex-edge balancing gets more and more challenging when the partition goes smaller, from nodes to sockets, then to cores. With smaller partitions, there are fewer flexibilities for tuning the α parameter to achieve inter-core load balance, especially for graphs with high per-vertex degree variances.

Leveraging shared memory not available to inter-node load balancing, Gemini employs a fine-grained work-stealing scheduler for intra-node edge processing. While the per-socket edge processing work is preliminarily partitioned with a locality-aware balanced manner across all the cores as a starting point, each thread only grabs a small *mini-chunk* of vertices to process (`signal/slot`) during the OpenMP parallel region. Again, due to our chunk-based partitioning scheme, this refinement retains contiguous processing, and promotes efficient cache utilization and message batching. Bookkeeping is also easy, as it only requires one counter per core to mark the current mini-chunk's starting offset, shared across threads and accessed through atomic operations. The default Gemini setting of mini-chunk size is 64 vertices, as used in our experiments.

Each thread first tries to finish its own per-core partition, then starts to steal mini-chunks from other threads' partitions. Compared with finely interleaved mini-chunk distribution from the beginning, this enhances memory access by taking advantage of cache prefetching. Also, this delays contention involved in atomic additions on the shared per-core counters to the epilogue of the whole computation. At that point, the cost is clearly offset by improved inter-core load balance.

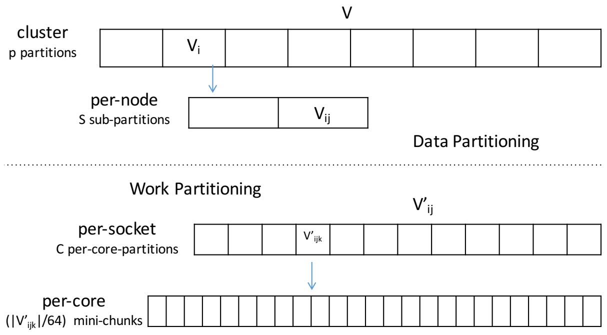


Figure 8: Hierarchical view of Gemini's chunking

Finally, we summarize Gemini's multi-level chunk-based partitioning in Figure 8, all the way from node-level, to socket-level, core-level, and finally to the mini-chunk granularity for inter-core work stealing. The illustration depicts the partitioning scenario in our actual test cluster with 8 nodes, 2 sockets per node, 12 cores per socket, and 64 vertices per mini-chunk. As shown here, such simple chunk-based partitioning can be refined in a hierarchical way, retaining access locality in edge processing continuously.

6 Implementation

Gemini is implemented in around 2,800 lines of C++ code, using MPI for inter-process communication and `libnuma` for NUMA-aware memory allocation. Below we discuss selected implementation details.

Graph Loading: When Gemini loads a graph from input file, each node reads its assigned contiguous portion in parallel. Edges are loaded sequentially into an edge buffer in batches, where they undergo an initial pass. Compared to common practice in existing systems, this reduces the memory consumption of the loading phase significantly, making it possible to load graphs whose scales approach the aggregate memory capacity of the whole cluster. For symmetric graphs, Gemini only stores the graph topology data of one mode, as sparse mode edges in this case are equivalent to the dense mode ones.

Graph Partitioning: When loading edges, each node calculates the local degree of each vertex. Next, an `AllReduce` operation collects such degree information for chunking the vertex set as discussed in Section 4. Each node can then determine the cuts locally without communication. Edges are then re-loaded from file and distributed to target nodes accordingly for constructing local subgraphs.

Memory Allocation: All nodes share the the node-level partitioning boundaries for inter-node message passing, while the socket-level sub-partition information is kept node-private. Each node allocates entire vertex arrays in shared memory. However, Gemini only touches data within its own vertex chunk, splitting the per-node vertex partition and placing the sub-chunks on corresponding sockets. The sub-partitioned graph topology datasets, namely edges and vertex indices, also adopts NUMA-aware allocation to promote localized memory accesses.

Mode Selection: Gemini follows Ligra's mode switching mechanism. For each `ProcessEdges` operation, Gemini first invokes an internal operation (defined via its `ProcessVertices` interface) to get the number of active edges, then determines the mode to use for the coming interation of processing.

Parallel Processing: When the program initializes, Gemini pins each OpenMP thread to specific sockets to prevent thread migration. For work-stealing, each thread maintains its status (`WORKING` or `STEALING`), current mini-chunk's start offset, and the pre-computed end offset, which are accessible to other threads and allocated in a NUMA-aware aligned manner to avoid false-sharing and unnecessary remote memory accesses (which should only happen in stealing stages). Each thread starts working from its own partition, changes the status when finished, and tries to steal work from threads with higher ranks in a cyclic manner. Concurrency control is via OpenMP's implicit synchronization mechanisms.

Message Passing: Gemini runs one process on each node, using MPI for inter-node message passing. At the inter-socket level, each socket produces/consumes messages through per-socket send and receive buffers in shared memory to avoid extra memory copies and perform NUMA-aware message batching.

7 Evaluation

We evaluate Gemini on the 8-node cluster, whose specifications are given in Section 2, running CentOS 7.2.1511. Intel ICPC 16.0.1 is used for compilation.

The graph datasets used for evaluation are shown in Table 2. Our evaluation uses five representative graph analytics applications: PageRank (*PR*), connected components (*CC*[1]), single source shortest paths (*SSSP*[2]), breadth first search (*BFS*), and betweenness centrality (*BC*). For comparison, we also evaluated state-of-the-art distributed graph processing systems, including PowerGraph (v2.2), GraphX (v2.0.0), and PowerLyra (v1.0), as well as shared-memory Ligra (20160826) and Galois (v2.2.1). For each system, we make our best effort to optimize the performance on every graph by carefully tuning the parameters, such as the partitioning method, the number of partitions, JVM options (for GraphX), the used algorithm (for Galois), etc. To get stable performance, we run *PR* for 20 iterations, and run *CC*, *SSSP*, *BFS*, and *BC* till convergence. The execution time is reported as elapsed time for executing the above graph algorithms (average of 5 runs) and does not include loading or partitioning time.

Graph	$\|V\|$	$\|E\|$
enwiki-2013	4,206,785	101,355,853
twitter-2010	41,652,230	1,468,365,182
uk-2007-05	105,896,555	3,738,733,648
weibo-2013	72,393,453	6,431,150,494
clueweb-12	978,408,098	42,574,107,469

Table 2: Graph datasets [5, 6, 7, 20] used in evaluation.

7.1 Overall Performance

As Gemini aims to provide scalability on top of efficiency, to understand the introduced overhead, we first take a zoom-in view of its single-node performance, using the five applications running on the *twitter-2010* graph. Here, instead of using distributed graph-parallel systems, we compare Gemini with two state-of-the-art shared-memory systems, Ligra and Galois, which we have verified to have superior performance compared

[1]Gemini makes the input graphs undirected when computing *CC*.

[2]A random weight between 0 and 100 is assigned to each edge.

with single-node executions of all the aforementioned distributed systems.

Application	Ligra	Galois	Gemini
PR	21.2	19.3	**12.7**
CC	6.51	**3.59***	4.93
SSSP	**2.81**	3.33	3.29
BFS	**0.347**	0.528	0.468
BC	2.45	3.94*	**1.88**

Table 3: 1-node runtime (in seconds) on input graph *twitter-2010*. The best times are marked in bold. "*" indicates where different algorithm is adopted (i.e. union-find for *CC* and asynchronous for *BC*).

Table 3 presents the performance of evaluated systems. Though with communication complexity designed for distributed execution, Gemini outperforms Ligra and Galois for *PR* and *BC*, and ranks the second for *CC*, *SSSP*, and *BFS*. With the use of NUMA-aware sub-partitioning, Gemini benefits from faster memory access and higher LLC utilization in edge processing, thanks to significantly reduced visits to remote memory. Unlike the NUMA-oblivious access patterns of Ligra and Galois, Gemini's threads only visit remote memory for work-stealing and message-passing.

Meanwhile, the distributed design inevitably brings additional overhead. The messaging abstraction (*i.e.*, batched messages produced by signals and consumed by slots) introduces extra memory accesses. This creates a major performance constraint for less computation-intensive applications like *BFS*, where the algorithm does little computation while the numbers of both visited edges and generated messages are in the order of $O(\Sigma|V_i|)$.[3] In contrast, most other applications access *all* adjacent edges of each active vertex, creating edge processing cost proportional to the number of active edges and sufficient to mask the message generation overhead.

Also, vertex state propagation in shared-memory systems employs direct access to the *latest* vertex states, while Gemini's BSP-based communication mechanism can only fetch the neighboring states through message passing in a super-step granularity. Therefore, its vertex state propagation lags behind that of shared-memory systems, forcing Gemini to run more iterations than Ligra and Galois for label-propagation-style applications like *CC* and *SSSP*.

Overall, with a relatively low cost paid to support distributed execution, Gemini can process much larger graphs by scaling out to more nodes and to work quite efficiently on single-node multi-core machines, allowing it to handle diverse application-platform combinations.

[3]In BFS's dense mode, edge processing at a vertex completes as soon as it successfully "pulls" from any of its neighbors [4].

Graph	PowerG.	GraphX	PowerL.	Gemini	Speedup (×times)
PR					
enwiki-2013	9.05	30.4	7.27	0.484	15.0
twitter-2010	40.3	216	26.9	3.02	8.91
uk-2007-05	64.9	416	58.9	1.48	39.8
weibo-2013	117	-	100	8.86	11.3
clueweb-12	-	-	-	31.1	n/a
CC					
enwiki-2013	4.61	16.5	5.02	0.237	19.5
twitter-2010	29.1	104	22.0	1.22	18.0
uk-2007-05	72.1	-	63.4	1.76	36.0
weibo-2013	56.5	-	58.6	2.62	21.6
clueweb-12	-	-	-	25.7	n/a
SSSP					
enwiki-2013	16.5	151	17.1	0.514	32.1
twitter-2010	12.5	108	10.8	1.15	9.39
uk-2007-05	117	-	143	3.45	33.9
weibo-2013	63.2	-	60.6	4.24	14.3
clueweb-12	-	-	-	56.9	n/a
GEOMEAN					19.1

Table 4: 8-node runtime (in seconds) and improvement of Gemini over the best of other systems. "-" indicates failed execution.

Table 4 reports the 8-node performance of PowerGraph, GraphX, PowerLyra, and Gemini, running *PR*, *CC*, and *SSSP* on all the tested graphs (*BFS* and *BC* results are omitted as their implementations are absent in other evaluated systems). The results show that Gemini outperforms the fastest of other systems in all cases significantly (19.1× on average), with up to 39.8× for *PR* on the *uk-2007-05* graph. For the *clueweb-12* graph with more than 42 billion edges, Gemini is able to complete *PR*, *CC*, and *SSSP* in 31.1, 25.7, and 56.9 seconds respectively on the 8-node cluster while *all* other systems fail to finish due to excessive memory consumption.

Graph	Raw	PowerGraph	Gemini
enwiki-2013	0.755	13.1	4.02
twitter-2010	10.9	138	32.1
uk-2007-05	27.8	322	73.1
weibo-2013	47.9	561	97.5
clueweb-12	318	-	597

Table 5: Peak 8-node memory consumption (in GB). "-" indicates incompletion due to running out of memory.

The performance gain mostly comes from the largely reduced distributed overhead. Table 5 compares the memory consumption of PowerGraph and Gemini. The raw graph size (with each edge in two 32-bit integers) is also presented for reference. PowerGraph needs memory more than 10× the raw size of a graph to process it. The larger memory footprint brings more instructions and memory accesses, and lowers the cache efficiency.

In contrast, while Gemini needs to store two copies of edges (in CSR and CSC respectively) due to its dual-mode propagation, the actual memory required is well controlled. Especially, the relative space overhead decreases for larger graphs (*e.g.*, within 2× of the raw size for *clueweb-12*). Gemini's abstraction (chunk-based partitioning scheme, plus the sparse-dense signal-slot processing model) adds very little overhead to the overall system and preserves (or enhances when more nodes are used) access locality present in the original graph. The co-scheduling mechanism hides the communication cost effectively under the high-speed Infiniband network. Locality-aware chunking and fine-grained work-stealing further improves inter-node and intra-node load balance. These optimizations together enable Gemini to provide scalability on top of efficiency.

7.2 Scalability

Next, we examine the scalability of Gemini, starting from intra-node evaluation using 1 to 24 cores to run *PR* on the *twitter-2010* graph (Figure 9). Overall the scalability is quite decent, achieving speedup of 1.9, 3.7, and 6.8 at 2, 4, and 8 cores, respectively. As expected, as more cores are used, inter-core load balancing becomes more challenging, synchronization cost becomes more visible, and memory bandwidth/LLC contention becomes intensified. Still, Gemini is able to achieve a speedup of 9.4 at 12 cores and 15.5 at 24 cores.

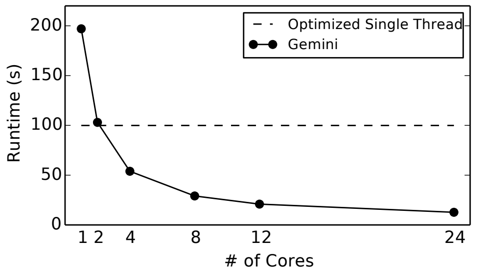


Figure 9: Intra-node scalability (*PR* on *twitter-2010*)

To further evaluate Gemini's computation efficiency, we compare it with the optimized single-thread implementation (which sorts edges in a Hilbert curve order [33]), shown as the dashed horizontal line in Figure 9. Using the COST metric (i.e. how many cores a parallel/distributed solution needs to outperform the optimized single-thread implementation), Gemini's number is 3, which is lower than those of other systems measured [33], though Gemini's 2-core execution time is only 3.1% higher than the optimized single-thread implementation. Considering Gemini's distributed nature, a COST close to 2 illustrates its optimized computation efficiency and lightweight distributed execution overhead.

Figure 10 shows the inter-node scalability results, comparing Gemini with PowerLyra, which we found to have the best performance and scalability for our test cases among existing open-source systems. Due to its higher memory consumption, PowerLyra is not able to complete in several test cases, as indicated by the missing data points. All results are normalized to Gemini's best execution time of the test case in question. It shows that though focused on computation optimization, Gemini is able to deliver inter-node scalability very similar to that by PowerLyra, approaching linear speedup with large graphs (*weibo-2013*). With the smallest graph (*enwiki-2013*), as expected, the scalability is poor for both systems as communication time dominates the execution.

For *twitter-2010*, Gemini has poor scaling after 4 nodes, mainly due to the emerging bottleneck from vertex indices access and message production/consumption. This is confirmed by the change of subgraph dimensions shown in Table 6: when more nodes are used, both $|E_i|$ and $|V_i|$ scales down perfectly, reducing edge processing cost. The vertex set including mirrors, V_i', however, does not shrink accordingly, making its processing cost increasingly significant.

$p \cdot s$	T_{PR} (s)	$\Sigma\|V_i\|/(p \cdot s)$	$\Sigma\|E_i\|/(p \cdot s)$	$\Sigma\|V_i'\|/(p \cdot s)$
$1 \cdot 2$	12.7	20.8M	734M	27.6M
$2 \cdot 2$	7.01	10.4M	367M	19.6M
$4 \cdot 2$	3.88	5.21M	184M	13.5M
$8 \cdot 2$	3.02	2.60M	91.8M	10.5M

Table 6: Subgraph sizes with growing cluster size

7.3 Design Choices

Below we evaluate the performance impact of several major design choices in Gemini. Though it is tempting to find out the relative significance among these optimizations themselves, we have found it hard to compare the contribution of individual techniques, as they often assist each other (such as chunk-based partitioning and intra-node work-stealing). In addition, when we incrementally add these optimizations to a baseline system, the apparent gains measured highly depend on the order used in such compounding. Therefore we present and discuss the advantages of individual design decisions, where results do not indicate their relative strength.

7.3.1 Adaptive Sparse-Dense Dual Mode

Adaptive switching between sparse and dense modes according to the density of active edges improves the performance of Gemini significantly. We propose an exper-

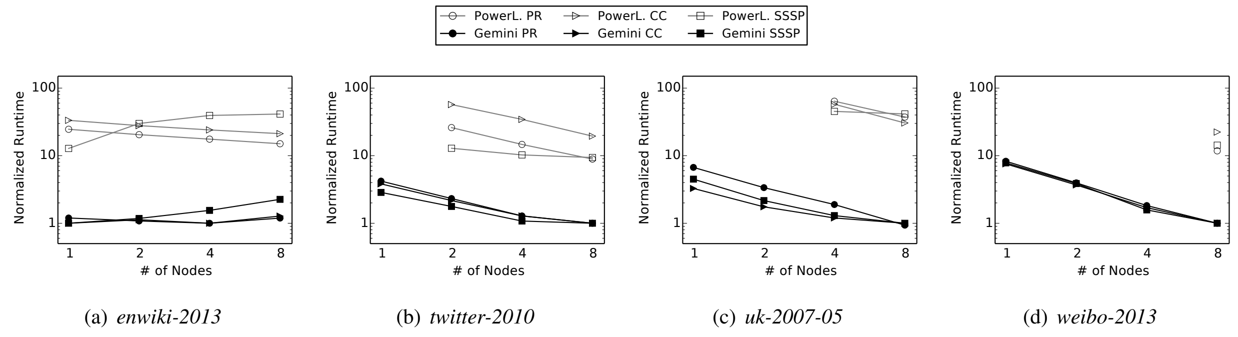


Figure 10: Inter-node scalability of PowerLyra and Gemini

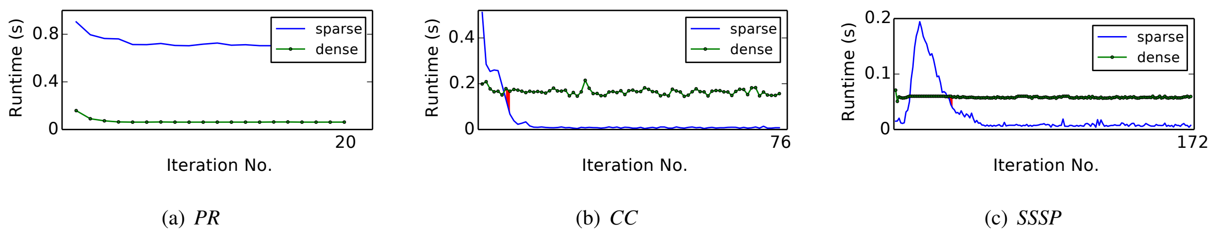


Figure 11: Gemini's per-iteration runtime in sparse and dense modes (*uk-2007-05*). Red regions indicate iterations where Gemini's adaptive engine chooses sub-optimal modes.

iment by forcing Gemini to run under the two modes for each iteration respectively to illustrate the necessities of the dual mode abstraction.

As shown in Figure 11, the performance gap between sparse and dense modes is quite significant, for all three applications. For PR, the dense mode consistently outperforms the sparse one. For CC, the dense mode performs better at the first few iterations when most of the vertices remain active, while the sparse mode is more effective when more vertices reach convergence. For SSSP, the sparse mode outperforms the dense mode in most iterations, except in a stretch of iterations where many vertices get updated. Gemini is able to adopt the better mode in most iterations, except 2 out of 76 for CC and 5 out of 172 iterations for SSSP. These "mis-predictions" are slightly sub-optimal as they happen, as expected, around the intersection of the two modes' performance curves.

7.3.2 Chunk-Based Partitioning

Next, we examine the effectiveness of Gemini's chunk-based partitioning through an experiment comparing it against hash-based partitioning[4].

Figure 12 exhibits the performance of Gemini using these two partitioning methods on *twitter-2010* and *uk-2007-05*, sampled to represent social and web graphs, respectively. Gemini's chunk-based partitioning outperforms the hash-based solution for both graphs. The performance improvement is especially significant for the web graph *uk-2007-05*, with more than 5.44× speedup. The reason behind is the locality-preserving property of chunk-based partitioning. Hash-based partitioning, in contrast, loses the natural locality in the original graph. As a result, hash-based partitioning produces not only higher LLC miss rates, but also a large number of mirrors in each partition, higher communication costs, and more

[4]We integrate the hash-based scheme (assigning vertex x to partition $x\%p$) into Gemini by re-ordering vertices according to the hashing result before chunking them.

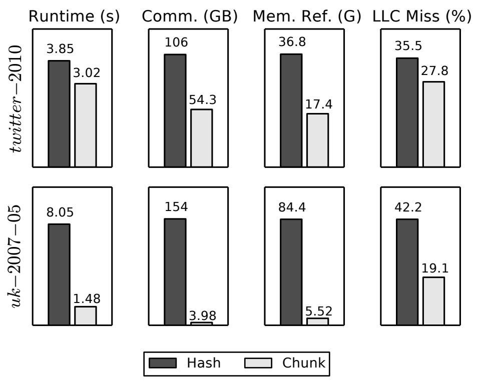


Figure 12: Hash- vs. chunk-based partitioning (PR on *twitter-2010* and *uk-2007-05*)

memory references, for master-mirror message passing and vertex index accesses.

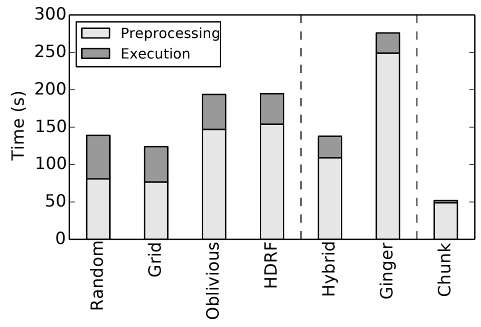


Figure 13: Preprocessing/execution time (*PR* on *twitter-2010*) with different partitioning schemes

Figure 13 shows the preprocessing time (loading plus partitioning) of different partitioning methods [12, 16, 24, 39] used by PowerGraph, PowerLyra, and Gemini on *twitter-2010*, with PR execution time given as reference. While it appears that preprocessing takes much longer than the algorithm execution itself, such preprocessing only poses a one-time cost, while the partitioned graph data can be re-used repeatedly by different applications or with different parameters.

NUMA-aware sub-partitioning plays another important role, as demonstrated by Figure 14 comparing sample Gemini performance with and without it. Without socket-level sub-partitioning, interleaved memory allocation leaves all accesses to the graph topology, vertex states, and message buffers distributed across both sockets. With socket-level sub-partitioning applied, instead, remote memory accesses are significantly trimmed, as they only happen when stealing work from or accessing messages produced by other sockets. The LLC miss rate and average memory access latency also decrease thanks to having per-socket vertex chunks.

7.3.3 Enhanced Vertex Index Representation

Table 7 presents the improvement brought by using bitmap assisted compressed sparse row and doubly compressed sparse column, with three applications on two input graphs. Compared with the original CSR/CSC formats, these enhanced data structures reduces memory consumption by 19-24%. They also eliminate many unnecessary memory accesses, bringing additional performance gain.

7.3.4 Load Balancing

Next, Table 8 portraits the benefit of Gemini's locality-aware chunking, by giving the number of owned vertices and dense mode edges in the *most time-consuming* partition. Compared with alternatives that aim at balancing vertex or edge counts, Gemini improves load balance by considering both vertex access locality and number of edges to be processed.

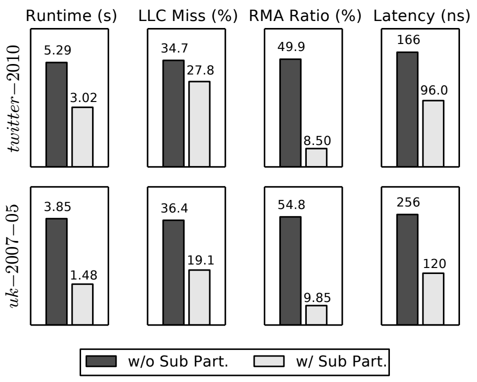


Figure 14: Impact of socket-level sub-partitioning (*PR* on *twitter-2010* and *uk-2007-05*)

Graph		*twitter-2010*	*uk-2007-05*
Mem. Reduction		19.4%	24.3%
	PR	1.25	2.76
Speedup	*CC*	1.11	1.30
	SSSP	1.14	1.98

Table 7: Impact of enhanced vertex index representation

Balanced By	Runtime (s)	$\lvert V_i\rvert$	$\lvert E_i^D\rvert$
$\lvert V_i\rvert$	5.51	5.21M	957M
$\lvert E_i^D\rvert$	3.95	18.1M	183M
$\alpha \cdot \lvert V_i\rvert + \lvert E_i^D\rvert$	3.02	0.926M	423M

Table 8: Impact of locality-aware chunking (*PR* on *twitter-2010*)

Finally, we evaluate the effect of Gemini's fine-grained work-stealing by measuring the improvement by three intra-node load balancing strategies. More specifically, we report the relative speedup of (1) static, pre-balanced per-core work partitions using our locality-aware chunking, (2) work-oblivious stealing, and (3) the integration of both (as adopted in Gemini), over the baseline using static scheduling. Table 9 lists the results. As expected, static core-level work partitioning is not enough to ensure effective multi-core utilization. Yet, pre-computed per-core work partitions do provide a good starting point when working jointly with work stealing.

Strategy	*twitter-2010*	*uk-2007-05*
Balanced partition	1.25	1.66
Stealing	1.55	1.93
Balanced partition + stealing	1.66	2.18

Table 9: *PR* speedup (over static scheduling) with different intra-node load balancing strategies

8 Related Work

We have discussed and evaluated several most closely related graph-parallel systems earlier in the paper. Here we give a brief summary of related categories of prior work.

A large number of graph-parallel systems [3, 10, 11, 12, 16, 17, 21, 22, 23, 26, 29, 30, 32, 36, 41, 42, 43, 44, 47, 49, 55, 56, 57, 59, 60] have been proposed for efficient processing of graphs with increasing scales. Gemini is inspired by prior systems in various aspects, but differs from them by taking a holistic view on system design toward single-node efficiency and multi-node scalability.

Push vs. Pull: Existing distributed graph processing systems either adopt a push-style [3, 26, 32, 43, 44] or a pull-style [11, 12, 16, 17, 23, 30] model, or provide both while used separately [13, 19, 22]. Recognizing the importance of a model that adaptively combines push and pull operators as shown by shared-memory approaches [4, 36, 47, 57], Gemini extends the hybrid push-pull model from shared-memory to distributed-memory settings through a signal-slot abstraction to decouple communication from computation, which is novel in the context of distributed graph processing.

Data Distribution: Traditional literature in graph partitioning [8, 12, 16, 24, 25, 30, 32, 39, 48] puts the main focus on reducing communication cost and load imbalance, without enough attention on the introduced overhead to distributed graph processing. Inspired by the implementation of several single-node graph processing systems [29, 42, 49, 57, 60], Gemini adopts a chunk-based partitioning scheme that enables a low-overhead scaling out design. When applying the chunking method in a distributed fashion, we address new challenges, including the sparsity in vertex indices, inter-node load imbalance, and intra-node NUMA issues, with further optimizations to accelerate computation.

Communication and Coordination: GraM [55] designs an efficient RDMA-based communication stack to overlap communication and computation for scalability. Gemini achieves similar goals by co-scheduling computation and communication tasks in a partition-oriented ring order, which is inspired by the implementation of collective operations in MPI [51], and can work effectively without the help of RDMA. PGX.D [22] highlights the importance of intra-node load balance to performance and proposes an edge chunking method. Gemini extends the idea by integrating chunk-based core-level work partitioning into a fine-grained work-stealing scheduler, which allows it to achieve better multi-core utilization.

There also exist many systems that focus on query processing [40, 46, 54], temporal analytics [13, 19, 27, 31], machine learning and data mining [50, 58], or more general tasks [34, 35, 45] on large-scale graphs. It would be interesting to explore how Gemini's computation-centric design could be applied to these systems.

9 Conclusion

In this work, we investigated computation-centric distributed graph processing, re-designing critical system components such as graph partitioning, graph representation and update propagation, task/message scheduling, and multi-level load balancing surrounding the theme of improving computation efficiency on modern multi-core cluster nodes. Our development and evaluation reveal that (1) effective system resource utilization relies on building low-overhead distributed designs upon optimized single-node computation efficiency, and (2) low-cost chunk-based partitioning preserving data locality across multiple levels of parallelism performs surprisingly well, and opens up many opportunities for subsequent optimizations throughout the system.

Meanwhile, through the evaluation of Gemini and other open-source graph processing systems, we have noticed that performance, scalability, and the location of bottleneck are highly dependent on the complex interaction between algorithms, input graphs, and underlying systems. Relative performance results comparing multiple alternative systems reported in papers (including this one) sometimes cannot be replicated with different platform configurations or input graphs. This also highlights the need of adaptive systems that customizes its decisions based on dynamic application, data, and platform behaviors.

Acknowledgments

We sincerely thank all the reviewers for their insightful comments and suggestions. We also thank Haibo Chen, Rong Chen, and Tej Chajed for their valuable feedback during our preparation of the final paper. This work is supported in part by the National Grand Fundamental Research 973 Program of China under Grant No. 2014CB340402, and the National Science Fund for Distinguished Young Scholars under Grant No. 61525202.

References

[1] `https://en.wikipedia.org/wiki/Signals_and_slots`.

[2] Apostolico, A., and Drovandi, G. Graph compression by bfs. *Algorithms 2*, 3 (2009), 1031–1044.

[3] Avery, C. Giraph: Large-scale graph processing infrastructure on hadoop. *Proceedings of the Hadoop Summit. Santa Clara* (2011).

[4] Beamer, S., Asanović, K., and Patterson, D. Direction-optimizing breadth-first search. *Scientific Programming 21*, 3-4 (2013), 137–148.

[5] Boldi, P., Rosa, M., Santini, M., and Vigna, S. Layered label propagation: A multiresolution coordinate-free ordering for compressing social networks. In *Proceedings of the 20th international conference on World wide web* (2011), ACM, pp. 587–596.

[6] Boldi, P., Santini, M., and Vigna, S. A large time-aware graph. *SIGIR Forum 42*, 2 (2008), 33–38.

[7] Boldi, P., and Vigna, S. The webgraph framework i: compression techniques. In *Proceedings of the 13th international conference on World Wide Web* (2004), ACM, pp. 595–602.

[8] Bourse, F., Lelarge, M., and Vojnovic, M. Balanced graph edge partition. In *Proceedings of the 20th ACM SIGKDD international conference on Knowledge discovery and data mining* (2014), ACM, pp. 1456–1465.

[9] Buluç, A., and Gilbert, J. R. On the representation and multiplication of hypersparse matrices. In *Parallel and Distributed Processing, 2008. IPDPS 2008. IEEE International Symposium on* (2008), IEEE, pp. 1–11.

[10] Buluç, A., and Gilbert, J. R. The combinatorial blas: Design, implementation, and applications. *International Journal of High Performance Computing Applications* (2011), 1094342011403516.

[11] Chen, R., Ding, X., Wang, P., Chen, H., Zang, B., and Guan, H. Computation and communication efficient graph processing with distributed immutable view. In *Proceedings of the 23rd international symposium on High-performance parallel and distributed computing* (2014), ACM, pp. 215–226.

[12] Chen, R., Shi, J., Chen, Y., and Chen, H. Powerlyra: Differentiated graph computation and partitioning on skewed graphs. In *Proceedings of the Tenth European Conference on Computer Systems* (2015), ACM, p. 1.

[13] 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* (2012), ACM, pp. 85–98.

[14] David, T., Guerraoui, R., and Trigonakis, V. Everything you always wanted to know about synchronization but were afraid to ask. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles* (2013), ACM, pp. 33–48.

[15] Faloutsos, M., Faloutsos, P., and Faloutsos, C. On power-law relationships of the internet topology. In *ACM SIGCOMM Computer Communication Review* (1999), vol. 29, ACM, pp. 251–262.

[16] Gonzalez, J. E., Low, Y., Gu, H., Bickson, D., and Guestrin, C. Powergraph: Distributed graph-parallel computation on natural graphs. In *OSDI* (2012), vol. 12, p. 2.

[17] Gonzalez, J. E., Xin, R. S., Dave, A., Crankshaw, D., Franklin, M. J., and Stoica, I. Graphx: Graph processing in a distributed dataflow framework.

[18] Gregor, D., and Lumsdaine, A. The parallel bgl: A generic library for distributed graph computations.

[19] 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* (2014), ACM, p. 1.

[20] Han, W., Zhu, X., Zhu, Z., Chen, W., Zheng, W., and Lu, J. Weibo, and a tale of two worlds. In *Proceedings of the 2015 IEEE/ACM International Conference on Advances in Social Networks Analysis and Mining 2015* (2015), ACM, pp. 121–128.

[21] Han, W.-S., Lee, S., Park, K., Lee, J.-H., Kim, M.-S., Kim, J., and Yu, H. Turbograph: a fast parallel graph engine handling billion-scale graphs in a single pc. In *Proceedings of the 19th ACM SIGKDD international conference on Knowledge discovery and data mining* (2013), ACM, pp. 77–85.

[22] Hong, S., Depner, S., Manhardt, T., Van Der Lugt, J., Verstraaten, M., and Chafi, H. Pgx.d: A fast distributed graph processing engine. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis* (New York, NY, USA, 2015), SC '15, ACM, pp. 58:1–58:12.

[23] Hoque, I., and Gupta, I. Lfgraph: Simple and fast distributed graph analytics. In *Proceedings of the First ACM SIGOPS Conference on Timely Results in Operating Systems* (2013), ACM, p. 9.

[24] Jain, N., Liao, G., and Willke, T. L. Graphbuilder: scalable graph etl framework. In *First International Workshop on Graph Data Management Experiences and Systems* (2013), ACM, p. 4.

[25] Karypis, G., and Kumar, V. Parallel multilevel series k-way partitioning scheme for irregular graphs. *Siam Review 41*, 2 (1999), 278–300.

[26] Khayyat, Z., Awara, K., Alonazi, A., Jamjoom, H., Williams, D., and Kalnis, P. Mizan: a system for dynamic load balancing in large-scale graph processing. In *Proceedings of the 8th ACM European Conference on Computer Systems* (2013), ACM, pp. 169–182.

[27] Khurana, U., and Deshpande, A. Efficient snapshot retrieval over historical graph data. In *Data Engineering (ICDE), 2013 IEEE 29th International Conference on* (2013), IEEE, pp. 997–1008.

[28] Kwak, H., Lee, C., Park, H., and Moon, S. What is Twitter, a social network or a news media? In *WWW '10: Proceedings of the 19th international conference on World wide web* (New York, NY, USA, 2010), ACM, pp. 591–600.

[29] Kyrola, A., Blelloch, G. E., and Guestrin, C. Graphchi: Large-scale graph computation on just a pc. In *OSDI* (2012), vol. 12, pp. 31–46.

[30] Low, Y., Bickson, D., Gonzalez, J., Guestrin, C., Kyrola, A., and Hellerstein, J. M. Distributed graphlab: a framework for machine learning and data mining in the cloud. *Proceedings of the VLDB Endowment 5*, 8 (2012), 716–727.

[31] MACKO, P., MARATHE, V. J., MARGO, D. W., AND SELTZER, M. I. Llama: Efficient graph analytics using large multiversioned arrays. In *Data Engineering (ICDE), 2015 IEEE 31st International Conference on* (2015), IEEE, pp. 363–374.

[32] 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* (2010), ACM, pp. 135–146.

[33] MCSHERRY, F., ISARD, M., AND MURRAY, D. G. Scalability! but at what cost? In *Proceedings of the 15th USENIX Conference on Hot Topics in Operating Systems* (Berkeley, CA, USA, 2015), HOTOS'15, USENIX Association, pp. 14–14.

[34] 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* (2013), ACM, pp. 439–455.

[35] NELSON, J., HOLT, B., MYERS, B., BRIGGS, P., CEZE, L., KAHAN, S., AND OSKIN, M. Latency-tolerant software distributed shared memory. In *2015 USENIX Annual Technical Conference (USENIX ATC 15)* (2015), pp. 291–305.

[36] NGUYEN, D., LENHARTH, A., AND PINGALI, K. A lightweight infrastructure for graph analytics. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles* (New York, NY, USA, 2013), SOSP '13, ACM, pp. 456–471.

[37] OUSTERHOUT, K., RASTI, R., RATNASAMY, S., SHENKER, S., AND CHUN, B.-G. Making sense of performance in data analytics frameworks. In *12th USENIX Symposium on Networked Systems Design and Implementation (NSDI 15)* (2015), pp. 293–307.

[38] PAGE, L., BRIN, S., MOTWANI, R., AND WINOGRAD, T. The pagerank citation ranking: Bringing order to the web.

[39] PETRONI, F., QUERZONI, L., DAUDJEE, K., KAMALI, S., AND IACOBONI, G. Hdrf: Stream-based partitioning for power-law graphs.

[40] QUAMAR, A., DESHPANDE, A., AND LIN, J. Nscale: neighborhood-centric analytics on large graphs. *Proceedings of the VLDB Endowment 7*, 13 (2014), 1673–1676.

[41] ROY, A., BINDSCHAEDLER, L., MALICEVIC, J., AND ZWAENEPOEL, W. Chaos: Scale-out graph processing from secondary storage. In *Proceedings of the 25th Symposium on Operating Systems Principles* (2015), ACM, pp. 410–424.

[42] 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* (2013), ACM, pp. 472–488.

[43] SALIHOGLU, S., AND WIDOM, J. Gps: A graph processing system. In *Proceedings of the 25th International Conference on Scientific and Statistical Database Management* (2013), ACM, p. 22.

[44] SEO, S., YOON, E. J., KIM, J., JIN, S., KIM, J.-S., AND MAENG, S. Hama: An efficient matrix computation with the mapreduce framework. In *Cloud Computing Technology and Science (CloudCom), 2010 IEEE Second International Conference on* (2010), IEEE, pp. 721–726.

[45] SHAO, B., WANG, H., AND LI, Y. Trinity: A distributed graph engine on a memory cloud. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data* (2013), ACM, pp. 505–516.

[46] SHI, J., YAO, Y., CHEN, R., CHEN, H., AND LI, F. Fast and concurrent rdf queries with rdma-based distributed graph exploration. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)* (Savannah, GA, Nov. 2016), USENIX Association.

[47] SHUN, J., AND BLELLOCH, G. E. Ligra: a lightweight graph processing framework for shared memory. In *ACM SIGPLAN Notices* (2013), vol. 48, ACM, pp. 135–146.

[48] STANTON, I., AND KLIOT, G. Streaming graph partitioning for large distributed graphs. In *Proceedings of the 18th ACM SIGKDD international conference on Knowledge discovery and data mining* (2012), ACM, pp. 1222–1230.

[49] SUNDARAM, N., SATISH, N., PATWARY, M. M. A., DULLOOR, S. R., ANDERSON, M. J., VADLAMUDI, S. G., DAS, D., AND DUBEY, P. Graphmat: High performance graph analytics made productive. *Proc. VLDB Endow. 8*, 11 (July 2015), 1214–1225.

[50] TEIXEIRA, C. H., FONSECA, A. J., SERAFINI, M., SIGANOS, G., ZAKI, M. J., AND ABOULNAGA, A. Arabesque: a system for distributed graph mining. In *Proceedings of the 25th Symposium on Operating Systems Principles* (2015), ACM, pp. 425–440.

[51] THAKUR, R., RABENSEIFNER, R., AND GROPP, W. Optimization of collective communication operations in mpich. *International Journal of High Performance Computing Applications 19*, 1 (2005), 49–66.

[52] UGANDER, J., KARRER, B., BACKSTROM, L., AND MARLOW, C. The anatomy of the facebook social graph. *arXiv preprint arXiv:1111.4503* (2011).

[53] VALIANT, L. G. A bridging model for parallel computation. *Communications of the ACM 33*, 8 (1990), 103–111.

[54] WANG, K., XU, G., SU, Z., AND LIU, Y. D. Graphq: Graph query processing with abstraction refinementscalable and programmable analytics over very large graphs on a single pc. In *2015 USENIX Annual Technical Conference (USENIX ATC 15)* (2015), pp. 387–401.

[55] WU, M., YANG, F., XUE, J., XIAO, W., MIAO, Y., WEI, L., LIN, H., DAI, Y., AND ZHOU, L. G ra m: scaling graph computation to the trillions. In *Proceedings of the Sixth ACM Symposium on Cloud Computing* (2015), ACM, pp. 408–421.

[56] YUAN, P., ZHANG, W., XIE, C., JIN, H., LIU, L., AND LEE, K. Fast iterative graph computation: a path centric approach. In *High Performance Computing, Networking, Storage and Analysis, SC14: International Conference for* (2014), IEEE, pp. 401–412.

[57] ZHANG, K., CHEN, R., AND CHEN, H. Numa-aware graph-structured analytics. In *Proc. PPoPP* (2015).

[58] ZHANG, M., WU, Y., CHEN, K., QIAN, X., LI, X., AND ZHENG, W. Exploring the hidden dimension in graph processing. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)* (Savannah, GA, Nov. 2016), USENIX Association.

[59] ZHENG, D., MHEMBERE, D., BURNS, R., VOGELSTEIN, J., PRIEBE, C. E., AND SZALAY, A. S. Flashgraph: Processing billion-node graphs on an array of commodity ssds. In *13th USENIX Conference on File and Storage Technologies (FAST 15)* (Santa Clara, CA, Feb. 2015), pp. 45–58.

[60] ZHU, X., HAN, W., AND CHEN, W. Gridgraph: Large scale graph processing on a single machine using 2-level hierarchical partitioning. In *USENIX ATC* (2015).

Fast and Concurrent RDF Queries with RDMA-based Distributed Graph Exploration

Jiaxin Shi, Youyang Yao, Rong Chen, Haibo Chen
Institute of Parallel and Distributed Systems,
Shanghai Jiao Tong University

Feifei Li
School of Computing,
University of Utah

Contacts: {rongchen, haibochen}@sjtu.edu.cn

Abstract

Many public knowledge bases are represented and stored as RDF graphs, where users can issue structured queries on such graphs using SPARQL. With massive queries over large and constantly growing RDF data, it is imperative that an RDF graph store should provide low latency and high throughput for concurrent query processing. However, prior systems still experience high per-query latency over large datasets and most prior designs have poor resource utilization such that each query is processed in sequence.

We present Wukong[1], a distributed graph-based RDF store that leverages RDMA-based graph exploration to provide highly concurrent and low-latency queries over large data sets. Wukong is novel in three ways. First, Wukong provides an RDMA-friendly distributed key/-value store that provides differentiated encoding and fine-grained partitioning of graph data to reduce RDMA transfers. Second, Wukong leverages full-history pruning to avoid the cost of expensive final join operations, based on the observation that the cost of one-sided RDMA operations is largely oblivious to the payload size to a certain extent. Third, countering conventional wisdom of preferring migration of execution over data, Wukong seamlessly combines data migration for low latency and execution distribution for high throughput by leveraging the low latency and high throughput of one-sided RDMA operations, and proposes a worker-obliger model for efficient load balancing.

Evaluation on a 6-node RDMA-capable cluster shows that Wukong significantly outperforms state-of-the-art systems like TriAD and Trinity.RDF for both latency and throughput, usually at the scale of orders of magnitude.

[1]Short for Sun Wukong, who is known as the Monkey King and is a main character in the Chinese classical novel "Journey to the West". Since Wukong is known for his extremely fast speed (21,675 kilometers in one somersault) and the ability to fork himself to do massive multi-tasking, we term our system as Wukong. The source code and a brief instruction on how to install Wukong is available at `http://ipads.se.sjtu.edu.cn/projects/wukong`.

1 Introduction

Many large datasets are continuously published using the Resource Description Framework (RDF) format, which represents a dataset as a set of $\langle subject, predicate, object \rangle$ triples that form a directed and labeled graph. Examples include Google's knowledge graph [20] and a number of public knowledge bases including DBpedia [1], Probase [51], PubChem-RDF [32] and Bio2RDF [7]. There are also a number of public and commercial websites like Google and Bing providing online queries through SPARQL[2] to such datasets.

With the increasing scale of RDF datasets and the growing number of queries per second, it is highly demanding that an RDF store provides low latency and high throughput over highly concurrent queries. In response, much recent research has been devoted to developing scalable and high performance systems to index RDF data and process SPARQL queries. Early RDF stores like RDF-3X [33], SW-Store [8], HexaStore [49] usually use a centralized design, while later designs such as TriAD [21], Trinity.RDF [54], H_2RDF [38] and SHARD [40] use a distributed store in response to the growing data sizes.

An RDF dataset is essentially a labeled, directed multigraph. Hence, it may be either stored as a set of triples as elements in relational tables (i.e., a triple store) [33, 21, 38, 53], or managed as a native graph (i.e., a graph store) [9, 58, 52, 54]. Prior work [54] shows that while using a triple store may enjoy query optimizations designed for relational database queries, query processing intensively relies on join operations over potentially large tables, which usually generates huge redundant intermediate data. Besides, using relational tables to store triples may limit the generality such that existing systems can hardly support general graph queries over RDF data such as reachability analysis and community detec-

[2]A recursive acronym for SPARQL Protocol and RDF Query Language.

tion [44].

In this paper, we describe Wukong, a distributed in-memory RDF store that provides low-latency, concurrent queries over large RDF datasets. To make it easy to scale out, Wukong follows a graph-based design by storing RDF triples as a native graph and leverages graph exploration to handle queries. Unlike prior graph-based RDF stores that are only designed to handle one query at a time, Wukong is also designed to provide high throughput such that it can handle hundreds of thousands of concurrent queries per second. The key techniques of Wukong are centered around using one-sided RDMA to provide fast and concurrent graph exploration.

RDMA-friendly Graph Model and Store (§4). Besides storing RDF triples as a graph by treating *object/subject* as vertices and *predicate* as edges, Wukong extends an RDF graph by introducing index vertices so that indexes are naturally parts of the graph. To partition and distribute data among multiple machines, Wukong applies a differentiated partition scheme [13] to embrace both locality (for normal vertices) and parallelism (for index vertices) during query processing. Based on the observation that RDF queries only touch a small subset of graph data (e.g., a subset of vertices and/or a subset of a vertex's data), Wukong further incorporates predicate-based finer-grained vertex decomposition and stores the decomposed graph data into a refined, RDMA-friendly distributed hashtable inherited from DrTM-KV [48] to reduce RDMA transfers.

RDMA-based Full-history Pruning (§5.2). Being aware of the cost-insensitivity of one-sided RDMA operations with respect to data size, Wukong leverages *full-history pruning* such that it can precisely prune unnecessary intermediate data. Consequently, Wukong can avoid the costly centralized final join on the results aggregated from multiple machines.

RDMA-based Query Distribution (§5.3). Depending on the selectivity and complexity of queries, Wukong decomposes a query into a sequence of sub-queries and handles multiple independent sub-queries simultaneously. For each sub-query, Wukong adopts an RDMA communication-aware mechanism: for small (selective) queries, it uses in-place execution that leverages one-sided RDMA read to fetch necessary data so that there is no need to move intermediate data; for large (non-selective) queries, it uses one-sided RDMA WRITE to distribute the query processing to all related machines. To prevent large queries from blocking small queries when handling concurrent queries, Wukong provides a latency-centric work stealing scheme (namely *worker-obliger model*) to dynamically oblige queries in straggling workers.

We have implemented Wukong and evaluated it on a 6-node cluster using a set of common RDF query benchmarks over a set of synthetic (e.g., LUBM and WSDTS) and real-life (e.g., DBPSB and YAGO2) datasets. Our experiment shows that Wukong provides orders of magnitude lower latency compared to state-of-the-art centralized (e.g., RDF-3X and BitMat) and distributed (e.g., TriAD and Trinity.RDF) systems. An evaluation using a mixture of queries on LUBM [3] shows that Wukong can achieve up to 269K queries per second on 6 machines with 0.80 milliseconds median latency.

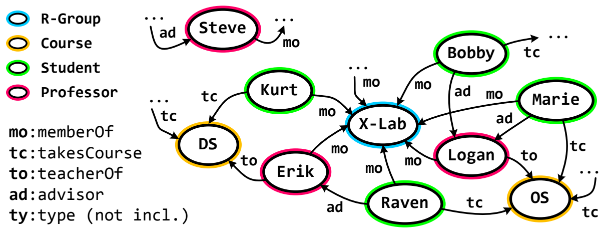


Fig. 1: An example RDF graph.

2 Background

2.1 RDF and SPARQL

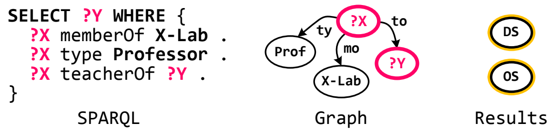
An RDF dataset is a graph (aka RDF graph) composed by triples, where a triple is formed by $\langle subject, predicate, object \rangle$. A triple can be regarded as a directed edge (*predicate*) connecting two vertices (from *subject* to *object*). Thus, an RDF graph can be alternatively viewed as a directed graph $G = (V, E)$, where V is the collection of all vertices (subjects and objects), and E is the collection of all edges, which are categorized by their labels (predicates). W3C has provided a set of unified vocabularies (as part of the RDF standard) to encode the rich semantics, where the rdfs:type predicate (or type for short) provides a classification of vertices of an RDF graph into different groups. As shown in Figure 1, a simplified sample RDF graph of LUBM dataset [3], the entity Steve has type Professor[3], and there are four categories of edges linking entities, namely, memberOf (mo), takesCourse (tc), teacherOf (to), and advisor (ad).

SPARQL, a W3C recommendation, is the standard query language for RDF datasets. The most common type of SPARQL queries is as follows:

Q := SELECT RD WHERE GP

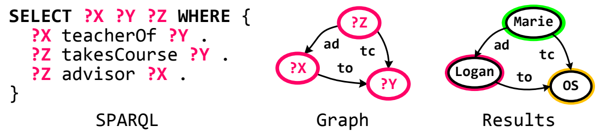
where, GP is a set of *triple patterns* and RD is a *result description*. Each triple pattern is of the form $\langle subject, predicate, object \rangle$, where each of the subject, predicate and object may denote either a *variable* or a *constant*. Given an RDF data graph G, the triple pattern GP searches on G for a set of subgraphs of G, each of which matches the graph pattern defined by GP (by binding pattern variables to values in the subgraph). The result description RD contains a subset of variables in the graph patterns.

[3]To save space, we use color circles to represent the type of entities.



```
SELECT ?Y WHERE {
  ?X memberOf X-Lab .
  ?X type Professor .
  ?X teacherOf ?Y .
}
```

Fig. 2: A SPARQL query (Q_1) on sample RDF graph.



```
SELECT ?X ?Y ?Z WHERE {
  ?X teacherOf ?Y .
  ?Z takesCourse ?Y .
  ?Z advisor ?X .
}
```

Fig. 3: A SPARQL query (Q_2) on sample RDF graph.

For example, as shown in Figure 2, the query Q_1 retrieves all objects that were taught (to) by a Professor who is a member (mo) of X-Lab. The query can also be graphically represented by a query graph, in which vertices represent the subjects and objects of the triple patterns; the black vertices represent constants, and the red vertices represent variables; The edges represent predicates in the required patterns (GP). The query results (?Y, described in RD) include DS and OS.

Difference from graph analytics. Readers might be curious about the relationship between RDF queries and graph analytics [28, 18, 31, 19, 13, 41, 55, 56], especially a recent design [50] used one-sided RDMA to implement message-passing primitives. However, there are several fundamental differences between RDF queries and graph analytics.

First, RDF queries are user-centric; thus minimizing the roundtrip latency is more important than maximizing network throughput. Second, RDF queries only touch a small subset of a graph instead of processing the entire graph, making it not worthwhile to dedicate all resources to run a single query. Third, graph-analytics is usually done in a batch-oriented manner in contrast to concurrently serving multiple RDF queries.

2.2 Existing Solutions

We then discuss two representative approaches adopted in existing state-of-the-art RDF systems.

Triple store and triple join: A majority of existing systems store and index RDF data as a set of *triples* in relational databases, and excessively leverage triple *join* operations to process SPARQL queries. Generally, query processing consists of two phases: Scan and Join. In the Scan phase, the RDF engine decomposes a SPARQL query into a set of triple patterns. For the query in Figure 2, the triple patterns are {?X memberOf X-Lab}, {?X type Professor} and {?X teacherOf ?Y}. For each triple pattern, it generates a temporary query table with bindings by scanning the triple store. In the Join phase, the query tables are joined to produce the final query results.

Some prior work [54] has summarized the inherent limitations of triple-store based approach. First, triple stores rely excessively on costly join operations, especially for distributed merge/hash-join. Second, the scan-join approach may generate large redundant intermediate results. Finally, while using redundant six primary SPO[4] *permutation indexes* [49] can accelerate scan operations, such indexes lead to heavy memory pressure.

Graph store and graph exploration: Instead of joining query tables, Trinity.RDF [49] stores RDF data in a native *graph* model on top of a distributed in-memory key/value store, and leverages fast *graph-exploration* strategy for query processing. It further adopts one-step pruning (i.e., the constraint in the immediately prior step) to reduce the intermediate results. As an example, considering Q_1 in Figure 2 over the data in Figure 1, after exploring the type of Professor for each member of X-Lab with respect to the data in Figure 1, we find that the possible binding for ?X is only Erik and Logan, and the rest of members are pruned.

However, the graph exploration in Trinity.RDF relies on a final centralized join to filter out non-matching results. For example, the query Q_2 in Figure 3 asks for advisors (?X), courses (?Y) and students (?Z) such that the advisor advises (ad) the student who also takes a course (tc) taught by (to) the advisor. After exploring all three triple patterns in Q_2 with respect to the data in Figure 1, the non-matching bindings, namely, Logan $\overrightarrow{to}$ OS, OS $\overleftarrow{tc}$ Raven and Raven $\overrightarrow{ad}$ Erik will not be pruned until a final join. Prior work [21, 37] indicates that the final join is a potential bottleneck, especially for queries with cycles and/or large intermediate results.

2.3 RDMA and Its Characteristics

Remote Direct Memory Access (RDMA) is a cross-node memory access technique with low-latency and low CPU overhead, due to complete bypassing of target OS kernel and/or CPU. RDMA provides both two-sided message passing interfaces like SEND/RECV Verbs as well as one-sided operations such as READ, WRITE and two atomic operations (fetch-and-add and compare-and-swap). As noted in prior work [30, 16, 48], one-sided operations are usually less disruptive than its two-sided counterparts due to no CPU involvement to the target machine. To minimize interference among multiple machines during query processing, we focus on one-sided RDMA operations in this paper. However, it should be straightforward to use two-sided RDMA operations in Wukong as well.

Figure 4(a) shows the throughput (in Kbps) of different communication primitives. RDMA undoubtedly achieves the highest throughput for all payload sizes, while the throughput of TCP/IP over IPoIB (IP over InfiniBand) or 10GbE approaches that of one-sided RDMA

[4]S, P and O stand for subject, predicate and object accordingly.

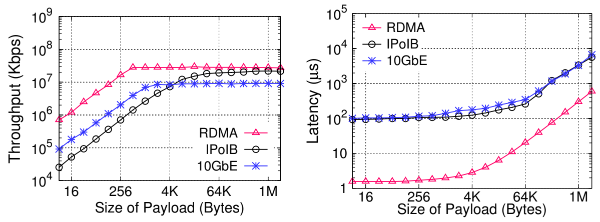


Fig. 4: (a) The throughput and (b) the latency of random reads using one-sided RDMA and TCP/IP with the increase of payload sizes.

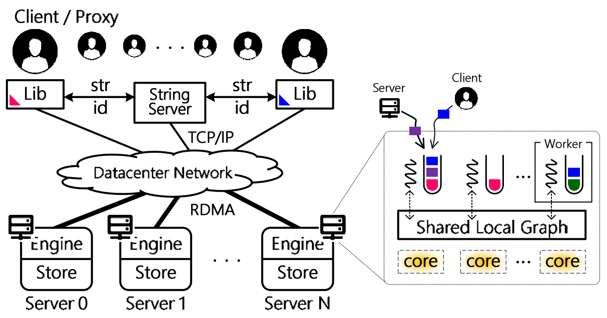


Fig. 5: The architecture overview of Wukong.

with the increase of payload sizes. For payload sizes larger than 4K bytes, the difference is limited to 4 times. In contrast, the gap of roundtrip latencies is always more than an order-of-magnitude, as shown in Figure 4(b). Therefore, it is imperative to leverage one-sided RDMA operations (i.e., READ and WRITE) to boost latency-oriented query processing. Further, an interesting feature is that the latency of RDMA is relatively *insensitive* to payload sizes, because small-sized requests cannot saturate the high-bandwidth network card[5]. For example, the latency only increases slightly (from 1.56μs to 2.25μs) even if the payload size increases 256X (from 8 bytes to 2K bytes).

3 Overview

Setting: Wukong assumes a cluster that is connected with a high-speed, low-latency network with RDMA features. Wukong targets SPARQL queries over a large volume of RDF data; it scales by partitioning an RDF graph into a large number of shards across multiple machines. Wukong may duplicate edges to make sure each machine contains a self-contained subgraph (e.g., no dangling edges) of the input RDF graph, for better locality. Wukong also creates index vertices to assist queries. In each machine, Wukong employs a worker-thread model by running n worker threads atop n cores; each worker thread executes a query at a time.

Architecture: An overview of Wukong's architecture is shown in Figure 5. Wukong follows a decentralized model on the server side, where each machine can directly serve clients' requests. Each client[6] contains a client library that parses SPARQL queries into a set of stored procedures, which are sent to the server side to handle the request. Alternatively, Wukong can also use a set of dedicated proxies to run the client-side library and balance client requests. Some sophisticated mechanisms like congestion control [57] and load balancing [36] can also be implemented at the proxy, which are beyond the scope of this paper. Moreover, to avoid sending and storing long strings and thus save network bandwidth and memory consumption, each string is first converted into a unique ID by the string server, similar to prior work [54, 21].

Each server consists of two separate layers: query engine and graph store. The query engine layer binds a worker thread on each core with a logical task queue to continuously handle requests from clients or other servers. The graph store layer adopts an RDMA-friendly key/value store over distributed hashtable to support a partitioned global address space. Each machine stores a partition of the RDF graph, which is shared by all of worker threads on the same machine.

Query processing: Wukong is designed to provide low-latency to multiple concurrent queries from clients. The client or the proxy decides which server a request will be first sent to according to the request types. For a query starting with a constant vertex, Wukong sends the request to the server holding the vertex. For a query starting with a set of vertices with a specific type or predicate, Wukong then sends the request to all replicas of the corresponding index vertex.

Wukong parses a query into an operator tree, the same as other systems. Each query may be represented as a chain of sub-queries. Each machine handles a sub-query and then dispatches the remaining sub-queries to other machines when necessary. A sub-query will be pushed into the task queue to be scheduled and executed asynchronously.

4 Graph-based RDF Data Modeling

This section provides a detailed description of the graph indexing, partitioning and storing strategies employed by Wukong, which are the basis to sequentially and concurrently process SPARQL queries on RDF data.

4.1 Graph Model and Indexes

Wukong uses a directed graph to model and store RDF data, where each vertex corresponds to an entity in an RDF triple (*subject* or *object*) and each edge is labeled as a *predicate* and points from subjects to objects. As SPARQL queries may rely on retrieving a set of subjects/object vertices connected by edges with certain predicates, we provide two kinds of *index* vertices to assist

[5]Note that this features also applies to other communication primitives (e.g., TCP/IP over IPoIB or 10GbE).

[6]The client may be not the end user but the front-end of Web service.

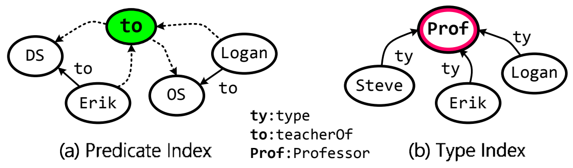


Fig. 6: Two types of index vertex of Wukong.

such queries, as shown in Figure 6. To avoid confusion, we use the *normal* vertex to refer to subjects and objects.

For the query pattern with a certain predicate, like {?Y teacherOf ?Z} (see Q_2 in Figure 3), we propose the *predicate* index (P-idx) to maintain all subjects and objects labeled with the particular predicate using its in and out edges respectively. The index vertex essentially serves as an inverted index from the predicate to the corresponding subjects or objects. For example, in Figure 6, a predicate index teacherOf (to) links to all normal vertices whose in-edges (DS and OS) or out-edges (Erik and Logan) contain the label to. This corresponds to the PSO and POS indexes in the triple store approaches.

Further, the special predicate type (ty) is used to group a set of subjects that belong to a certain type, like {?X type Prof} (see Q_1 in Figure 2). Therefore, we treat the objects of such predicate as the *type* index (T-idx), instead of providing a uniform but useless predicate index type to link all objects and subjects. For example, a type index Prof in Figure 6(b) maintains all normal vertices which are of the type of professors.

Unlike prior graph-based approaches that manage indexes using separate data structures, Wukong treats indexes as essential parts (vertices and edges) of an RDF graph and also takes into consideration the partitioning and storing of such indexes. This has two benefits. First, this eases query processing using graph exploration such that the graph exploration can directly start from an index vertex. Second, this makes it easy and efficient to distribute the indexes among multiple servers, as shown in the following sections.

4.2 Differentiated Graph Partitioning

One key step of supporting distributed query is partitioning a graph among multiple machines, while still preserving good access locality and enabling parallelism. We observe that complex queries usually involve a large number of vertices through a certain predicate or type, which should be executed on multiple machines to exploit parallelism.

Inspired by PowerLyra [13], Wukong adopts differentiated partitioning algorithms to normal and index vertices. One difference is that unlike PowerLyra, Wukong does not use the degrees to differentiate vertices, because an RDF query only navigates through a vertex and then routes to only a portion of its neighbors. Therefore, unlike graph analytics, a high-degree vertex in skewed

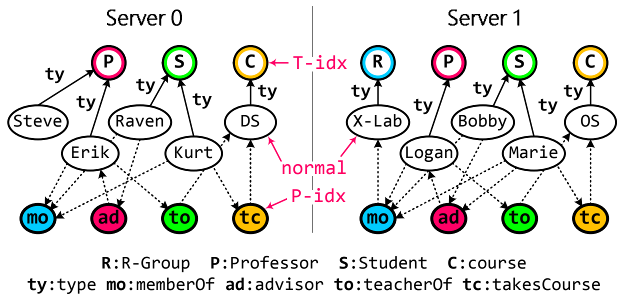


Fig. 7: A hybrid graph partitioning on two servers.

graphs does not necessarily incur significant imbalance for query processing, and it can be handled by fork-join execution appropriately (§ 5.3).

As shown in Figure 7, each normal vertex (e.g., DS) will be randomly assigned (i.e., by hashing the vertex ID) to only one machine with all of its edges (IDs of neighbors). Note that the edges linked to predicate index (i.e., dotted arrows) will not be included in the edge list of normal vertices, since there is no need to find a predicate index vertex via normal vertices and this can save plenty of memory. Different from a normal vertex, each index vertex (e.g., takesCourse and Course) will be split and replicated to multiple machines with edges linked to normal vertices on the same machine. This naturally distributes the indexes and their load among each machine.

4.3 RDMA-friendly Predicate-based Store

Similar to Trinity.RDF [54], Wukong uses a distributed key/value store to physically store the graph. However, unlike prior work that simply uses vertex ID (vid) as the key, and the in and out edge list (each element is a $\langle predicate, vid \rangle$ pair) as the value, Wukong uses a combination of the vertex ID (vid), predicate/type ID (p/tid) and in/out direction (d) as the key (in the form of $\langle vid, p/tid, d \rangle$), and the list of neighboring vertex IDs or predicate/type IDs as the value. The main observation is that an SPARQL query is usually concerned with querying upon partial neighboring vertices satisfying a particular predicate (e.g., X predicate ?Y). Therefore, missing the predicate and direction information in the key would lead to plenty of unnecessary computation cost and networking traffic. The finer-grained vertex decomposition using predicates also makes it possible to build local predicate indexing, which corresponds to the PSO and POS indexes in triple store approaches.

To uniformly store normal and index vertices and adapt differentiated partitioning strategies, Wukong separates the ID mapping for vertex ID (vid) and predicate/-type ID (p/tid). The ID 0 of vid (INDEX) is reserved for the index vertex, while the ID 0 and 1 of p/tid are reserved for the predicate and type indexes respectively. Figure 8 illustrates part of detailed cases on the sample graph. The key of normal vertex starts from a nonzero vid

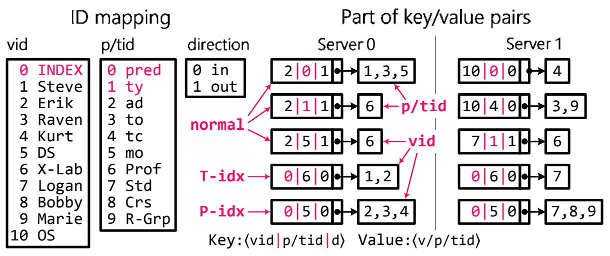


Fig. 8: The design of predicate-based key/value store.

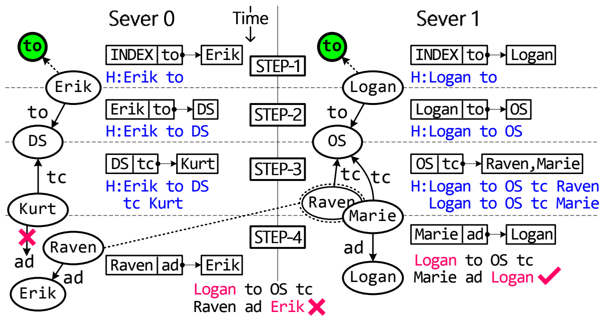


Fig. 9: A sample of execution flow on Wukong. The blue label H: shows the full history.

and relies on p/tid to distinguish different meanings of the value. The p/tid ID 0 and 1 represent the value as a list of predicate IDs and a type ID for the vertex respectively; otherwise the value is a list of normal vertices linked to the normal vertex with a certain predicate (p/tid). For example, the predicates labeled on out-edges of vertex Erik is represented as the key ⟨2|0|1⟩, and the value ⟨1,3,5⟩ means type, teacherOf and memberOf. While the type of vertex Erik is represented as the key ⟨2|1|1⟩, and the value ⟨6⟩ means Professor. The key of an index vertex always starts from a zero vid, and linked to a list of local normal vertices. For example, all subjects of the predicate memberOf on Server 0 (Erik, Raven and Kurt) and Server 1 (Logan, Bobby and Marie) are stored with the same key ⟨0|5|0⟩ but on different servers.

Finally, due to the goal of leveraging the advanced networking features such as RDMA, Wukong is built upon an RDMA-friendly distributed hashtable derived from DrTM-KV [48] and thus enjoys its nice features like RDMA-friendly cluster hashing and location-based cache. However, as the key/value store in Wukong is designed for query processing instead of transaction processing, we notably simplify the design by removing unnecessary metadata for checking consistency and supporting transactions. Likewise, other *symmetric* RDMA-friendly stores [16] can also work with Wukong to store RDF graph and support query processing (§5).

5 Query Processing

5.1 Basic Query Processing

An RDF query can be represented as a subgraph with free variables (i.e., not bound to specific subjects/objects yet). The goal of the query is to find bindings of specific subjects/objects to the free variables while respecting the subgraph pattern. However, it is well-known that using subgraph matching would be very costly due to the frequent yet costly joins [54]. Hence, like prior work [54], Wukong leverages graph exploration by walking the graph in specific orders according to each edge of the subgraph.

There are several cases for each edge in a graph query, depending on whether the subject, the predicate or the object is a free variable. For the common cases where the predicate is known but the subject/object are free variables, Wukong can leverage the predicate index to begin the graph exploration. Take Q_2 in Figure 3 as an example, which aims at querying advisors, courses and students such that the advisor advises the student who also takes a course taught by the advisor. The query forms a cyclic subgraph containing three free variables. Wukong chooses an order of exploration according to a cost-based approach with some heuristics.

As shown in Figure 9, Wukong starts exploration from the teacherOf predicate (to). Since Wukong extends the graph with predicate indexes, it can start exploration from the index vertex for teacherOf in each machine in parallel, whose neighbors contain Erik and Logan in each server accordingly. In Step2, Wukong combines Erik and Logan with teacherOf to form the key to get the corresponding courses, which are {Erik $\overrightarrow{to}$ DS} and {Logan $\overrightarrow{to}$ OS} accordingly. In Step3, Wukong continues to explore the graph from the course vertex for each tuple in parallel and tries to get all students that take the course. Thanks to the differentiated graph partitioning, there is no communication through Step1-3. In Step4, Wukong leverages the constraint information to filter out non-matching results to get the final result.

For (rare) cases where the predicate is unknown, Wukong starts graph exploration from a constant vertex (in cases where either subject or object is known) with a reserved p/tid 0 (pred). The value is the list of predicates associated with the vertex, and then Wukong iterates over them one by one. The remaining process is similar to those described above.

5.2 Full-history Pruning

Note that there could be tuples that should be filtered out during the graph exploration. For example, since there is no expected advisor predicate (ad) for Kurt, the related tuples should be filtered out to minimize redundant computation and communication. Further, in Step 4, as Raven's advisor is Erik instead of Logan, the graph exploration path also should be pruned as well.

Prior graph-exploration strategies [54] usually use a one-step pruning approach by leveraging the constraint in the immediately prior step to filter out unnecessary data (e.g., only DS and OS in Step 3). In the final step, it leverages a single machine to aggregate and conduct a final join over the results to filter out non-matching results. However, recent study [21, 37] found that, the final join can easily become the bottleneck of a query since all results need to be aggregated into a single machine for joining. Our experiment on LUBM [3] shows that some query spends more than 90% of execution time on the final join (details in §7.3).

Instead, Wukong adopts a *full-history pruning* approach such that Wukong passes the full exploration history to the next step within or across machines. The main observation is that, the cost of RDMA operations is insensitive to the payload size when it is smaller than a certain size (e.g., 2K bytes). Besides, the steps and variables in an RDF query are usually not many (i.e., less than 10), and each history item only contains subject/object/predicate IDs. Thus there won't be too much information carried even for the final few steps. Consequently, the cost remains low even passing more history information across machines. Further, improving the locality of graph exploration can also avoid additional network traffic from the full-history pruning.

As shown in Figure 9, Wukong passes {Erik $\overrightarrow{to}$}, {Erik $\overrightarrow{to}$ DS} and {Erik $\overrightarrow{to}$ DS $\overleftarrow{tc}$ Kurt} locally on Server 0 in each step; Kurt can be simply pruned without using history information due to no expected predicate (ad). Server 0 can leverage the full history ({Logan $\overrightarrow{to}$ OS $\overleftarrow{tc}$ Raven}) from Server 1 to prune Raven as Raven's advisor is not Logan.

As Wukong has the full history during graph exploration, there is no need of a final join to filter out non-matching results. Though it appears that Wukong may bring additional network traffic when fetching cross-machine history, the fact that Wukong can prune non-matching results early may save network traffic as well. For example, the query L1 on LUBM-10240 can benefit from early pruning to save about 96% network traffic (462MB vs. 18MB). Besides, many query histories are passed within a single machine and thus do not cause additional network traffic. In case the full history size is excessively large, Wukong can adaptively fall back to one-step pruning for the sub-query. However, we did not encounter such a case during our evaluation.

5.3 Migrating Execution or Data

During the graph exploration process, there will be different tradeoffs on whether migrating data or execution. Wukong provides *in-place* and *fork-join* executions accordingly. For a query step, if only a few vertices need to be fetched from remote machines, Wukong uses in-place execution mode that synchronously leverages one-sided RDMA READ to directly fetch vertices from remote machines, as shown in Figure 10(a). Using one-sided RDMA READ can enjoy the benefit of bypassing remote CPU and OS. For example, in Figure 9, Server 1 can directly read the advisor of Raven (i.e., Erik) by one RDMA READ, and locally generate ({Logan $\overrightarrow{to}$ OS $\overleftarrow{tc}$ Raven $\overrightarrow{ad}$ Erik}).

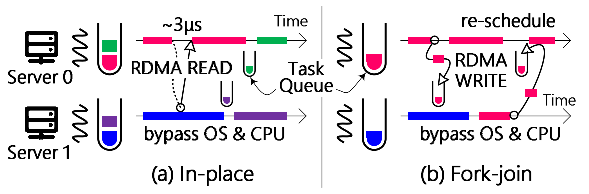


Fig. 10: A sample of (a) in-place and (b) fork-join execution.

For a query step, if many vertices may be fetched, Wukong leverages a *fork-join* execution mode that asynchronously splits the following query computation into multiple sub-queries running on remote machines. Wukong leverages one-sided RDMA WRITE to directly push a sub-query with full history into the task queue of a remote machine, as shown in Figure 10(b). This can also be done without bothering remote CPU and OS. For example, in Figure 9, Server 1 can send a sub-query with the full history ({Logan $\overrightarrow{to}$ OS $\overleftarrow{tc}$ Raven}) to Server 0. Server 0 will locally execute the sub-query to generate ({Logan $\overrightarrow{to}$ OS $\overleftarrow{tc}$ Raven $\overrightarrow{ad}$ Erik}). Note that, depending on the sub-query, the target machine may further do a fork-join operation to remote machines, forming a query tree. Each fork point then joins its forked sub-queries and returns the results to the parent fork point. In addition, all of sub-queries will be executed asynchronously without any global barrier and communication among worker threads. Even if two sub-queries access the same vertex, they are still independent due to working on different exploration paths.

Since the cost of RDMA operations is insensitive to the size of the payload, for each query step, Wukong decides on the execution mode at runtime according to the number of RDMA operations ($|N|$) for the next step. for the fork-join mode, $|N|$ is twice the number of servers; for the in-place mode, $|N|$ is equal to the number of required vertices. Each server will decide individually. Wukong simply uses a heuristic threshold according to the setting of cluster. Further, some vertices have a significant large number of edges with the same predicate, resulting in slower RDMA READ due to oversized payload. Wukong can label such vertices associated with the predicate to force the use of the fork-join mode when partitioning the RDF graph.

5.4 Concurrent Query Processing

Depending on the complexity and selectivity, the latency (i.e., execution time) of a query may vary significantly. For example, the latency differences among seven

```
int next = 1

OBLIGER()
  s = state[(tid+next)%N]
  q = NULL
  s.lock()
  if (s.cur == tid //reentry
    || s.end < now)
    s.cur = tid;
    s.end = now + T
    next++
    q = s.dequeue()
  s.unlock()
  return q

SELF()
  s = state[tid]
  s.lock()
  s.cur = tid
  s.end = now + T
  next = 1
  q = s.dequeue()
  s.unlock()
  return q

NEXT_QUERY()
  if (q = OBLIGER())
    return q
  return SELF()
```

Fig. 11: The pseudo-code of worker-obliger algorithm.

queries in LUBM [3] can reach around 3,000X (0.17ms and 516ms for L5 and L1 queries accordingly). Hence, dedicating an entire cluster for a single query, as done in prior approaches [54, 21], is not cost-effective.

Wukong is designed to handle a massive number of queries concurrently while trying to parallelize a single query to reduce the query latency. The difficulty is that, given the significantly varied query latencies, how to minimize inter-query interference while providing good utilization of resources, e.g., a lengthy query should not significantly extend the latency of a fast query.

The online sub-query decomposition and the dynamic execution mode switching serve as a keystone to support massive queries in parallel. Specifically, Wukong uses a private FIFO queue to schedule queries for each worker thread, which works well for small queries. However, if there is a lengthy query, it will monopolize the worker thread and impose queuing delays on the execution of small waiting queries. This will incur much higher latency than necessary. Worse even, a lengthy query with multi-threading enabled (Section 6) may monopolize the entire cluster.

The work stealing mechanism [10] is widely used to provide load balance in parallel systems, which allows tasks can be stolen from any queue of worker threads. However, the traditional algorithm is inefficient as the stolen tasks in Wukong are mostly sub-millisecond latency queries. Further, the unrestricted stealing among all workers may incur large overhead due to high contention.

To this end, Wukong uses a *worker-obliger* work stealing algorithm for multiple workers on each machine, as shown in Figure 11. Each worker is designated to oblige next few neighboring workers in case they are busy with processing a lengthy (sub-)query. After finishing a (sub-)query, a worker first checks a neighboring worker in turn if its (sub-)query has a timeout (i.e., `s.end < now`). If so, that worker might be handling a lengthy query and thus its following up queries may be delayed. In this case, this obliging worker steals one query from that worker's queue to process. After obliging its neighboring workers (until seeing a non-busy one), the worker

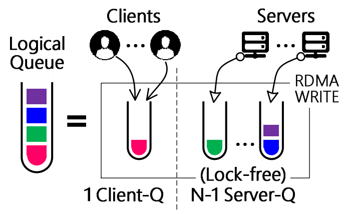


Fig. 12: The logical task queue in Wukong.

will then continue to handle its own queries by dequeuing from its own worker queue.

Note that, when all workers can handle their queries within a time threshold (i.e., `T`), each worker only needs to handle queries in its own queue. The checking code is also very lightweight and the state lock (i.e., `s.lock()`) won't be contended as there will only at most two workers (i.e., SELF and OBLIGER) may try to acquire the lock in usual. It could be possible that an obliger get stucked in handling a lengthy query for others; in this case, another worker may oblige this worker similarly.

6 Implementation

The Wukong prototype comprises around 6,000 lines of C++ code. It currently runs atop an RDMA-capable cluster. This section describes some implementation issues.

Task queues Wukong binds a worker thread on each core with a logical private task queue, which is used by both clients and worker threads on other servers to submit (sub-)queries. Wukong leverages RDMA operations (especially one-sided RDMA) to accelerate the communication among worker threads; however, the clients may still connect servers using general interconnects.

The logical queue per thread in Wukong consists of one client queue (Client-Q) and multiple server queues (Server-Q). For the client queue, Wukong follows traditional concurrent queue to serve the queries from many clients. But due to the lack of expressiveness of one-sided RDMA operations, implementing RDMA-based concurrent queue may incur large overhead. On the contrary, using separate task queues for each worker threads of each remote machine may exponentially increase the number of queues. Fortunately, we observe that there is no need to allow all worker threads on a remote machine sending queries to all local worker threads. To remedy this, Wukong only provides a one-to-one mapping between the work threads on different machines, as shown in Figure 12. This can avoid not only the burst of task queues but also complicated concurrent mechanisms.

Launching query To launch a query, the start point of a query can be a normal vertex (e.g., {?X memberOf X-Lab}) or a predicate or type index (e.g., {?X teacherOf ?Y}). Since the index vertex is replicated to multiple servers, Wukong allows the client library to send the same query to all servers such that the query can be dis-

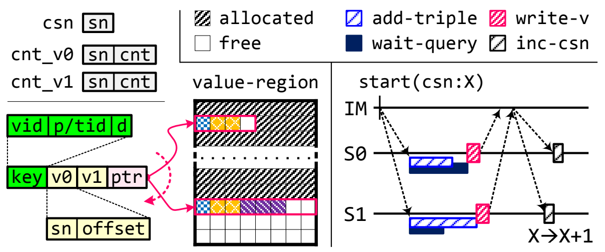


Fig. 13: The extension of graph store and the execution flow of injection for evolving RDF graphs.

tributed from the beginning. However, distributed execution may not be worthwhile for a low-degree index vertex. Therefore, Wukong will decide whether replicas of an index vertex need to process the query or not when partitioning the RDF graph. For low-degree index vertices, the master will process the query alone by aggregating data from replicas through one-sided RDMA READ, and the replicas will simply discard queries. For high-degree index vertices, both the master and replicas will individually process the query on local graph.

Multi-threading By default, Wukong processes a (sub-)query using only a single thread on each server. To reduce latency of a query, Wukong also allows running a time-consuming query with multiple threads on each server, at the requests of the client. A worker thread received the multi-threaded (MT) query will invite other worker threads on the same server to process the query in parallel. Wukong adopts a data-parallel approach to automatically parallelize the query after the first graph exploration. Each worker thread will individually process the query on a part of subgraph. Note that the maximum number of participants for a query is claimed by the client, but finally restricted by the MT threshold of the server.

Evolving graph While most prior RDF stores only support read-only queries, Wukong is also built with preliminary support to incrementally update the graph with concurrent queries. New triples will be periodically ingested to the RDF store, and all queries will run a consistent snapshot. Figure 13 illustrates three extensions to Wukong to support incremental update.

RDF Store. To support the dynamic increase of value, Wukong provides a buddy memory allocator. When the value space is full, the allocator will find a free value with double capacity, copy all data of the old value to the new one, and replace the pointer of the key using an atomic instruction. Further, to provide a consistent snapshot to above queries, each key should be extended with two versions (v0 and v1) that consist of its snapshot number and the offset within its value. The left part of Figure 13 illustrate the extension of RDF store.

Query processing. On each machine, there are two global reference counters (cnt_v0 and cnt_v1) to record

Table 1: A collection of real-life and synthetic datasets.

Dataset	#Triples	#Subjects	#Objects	#Predicates
LUBM-10240	1,410 M	222 M	165 M	17
WSDTS	109 M	5.2 M	9.8 M	86
DBPSB	15 M	0.3 M	5.2 M	14,128
YAGO2	190 M	10.5 M	54.0 M	99

the number of outstanding queries on two latest snapshots, and a current snapshot number (csn). Each query will first read the current snapshot number, and actively increase and decrease the corresponding counter before and after execution. The snapshot number of a query will be used to fetch a consistent version of all values and be inherited by all of its sub-queries.

RDF data injection. The added RDF triples in the new graph will be locally injected into all servers, which is coordinated by a single injection master (IM). Wukong performs the injection by executing the following steps. First, all triples are added in the background and remain invisible to concurrent queries. Meanwhile, all outstanding queries on the older snapshot (between v0 and v1) should be completed in advance. After they are done, each server will safely overwrite the older version within the keys by the new one and notify IM. When all servers are ready, IM will finally ask all servers to finish the injection of the new snapshot by atomically increasing the current snapshot number (csn) and the older global counter (between cnt_v0 and cnt_v1). The right part of Figure 13 shows the execution flow of the injection of the snapshot X+1 on two servers (S0 and S1).

7 Evaluation

7.1 Experimental Setup

Hardware configuration: All evaluations were conducted on a rack-scale cluster with 6 machines. Each machine has two 10-core Intel Xeon E5-2650 v3 processors and 64GB of DRAM. Each machine is equipped with two ConnectX-3 MCX353A 56Gbps InfiniBand NICs via PCIe 3.0 x8 connected to a Mellanox IS5025 40Gbps IB Switch, and an Intel X520 10GbE NIC connected to a Force10 S4810P 10GbE Switch. All machines run Ubuntu 14.04 with Mellanox OFED v3.0-2.0.1 stack.

In all experiments, we reserve two cores on each processor to generate requests for all machines to avoid the impact of networking between clients and servers as done in prior OLTP work [48, 17, 47, 46]. For a fair comparison, we measure the query execution time by excluding the cost of literal/ID mapping. All experimental results are the average of five runs.

Benchmarks: We use two synthetic and two real-life datasets, as shown in Table 1. The synthetic datasets are the Leigh University Benchmark (LUBM) [3] and the Waterloo SPARQL Diversity Test Suite (WSDTS) [5]. For LUBM, we generate 5 datasets with different sizes using the generator v1.7 in NT format. For queries, we

Table 2: The query performance (msec) on a single machine.

LUBM 2560	Wukong	TriAD	TriAD-SG (50K)	RDF-3X (mem)	BitMat (mem)
L1	752	621	3,315	2.3E5	abort
L2	120	149	221	4,494	36,256
L3	306	316	3,101	3,675	752
L4	0.19	3.38	3.34	2.2	55,451
L5	0.11	2.34	1.36	1.0	52
L6	0.56	20.7	6.06	37.5	487
L7	671	2,176	2,753	9,927	19,323
Geo. M	15.7	72.3	108	441	–

Table 3: The query performance (msec) on a 6-node cluster.

LUBM 10240	Wukong	TriAD	TriAD-SG (200K)	Trinity .RDF	SHARD
L1	516	2,110	1,422	12,648	19.7E6
L2	78	512	695	6,081	4.4E6
L3	203	1,252	1,225	8,735	12.9E6
L4	0.41	3.4	3.9	5	10.6E6
L5	0.17	3.1	4.5	4	4.2E6
L6	0.89	63	4.6	9	8.7E6
L7	464	10,055	11,572	31,214	12.0E6
Geo. M	16	190	141	450	9.1E6

use the benchmark queries published in Atre et al. [9], which were widely used by many distributed RDF systems [21, 54, 27]. WSDTS publishes a total of 20 queries in four categories. The real-life datasets are the DBpedia's SPARQL Benchmark (DBPSB) [1] and YAGO2 [6, 22]. For DBPSB, we choose 5 queries provided by its official website. YAGO2 is a semantic knowledge base, derived from Wikipedia, WordNet and GeoNames. We follow the queries defined in H_2RDF+ [37].

Comparing targets: We compare the query performance of Wukong against several state-of-the-art systems. 1) centralized systems: RDF-3X [33] and BitMat [9]; 2) distributed systems: TriAD [21], Trinity.RDF [54] and SHARD [40]. Since Trinity.RDF is not publicly available and TriAD reported superior performance over it, we only directly compare the results published in their paper [54] with the same workload. Except Wukong, all systems run over InfiniBand using IPoIB. We also enable string server for all systems to save memory consumption, reduce network bandwidth, and boost string matching.

7.2 Single Query Performance

We first study the performance of Wukong for a single query using the LUBM dataset.

For a fair comparison to centralized systems, we run Wukong and TriAD on a single machine and report the in-memory performance of RDF-3X and BitMat. As shown in Table 2[7], Wukong has significantly outperformed RDF-3X and BitMat by several orders of magnitude, due to fast graph exploration for simple queries and efficient multi-threading for complex queries. Note that L3 has an empty final result even with huge intermediate results and thus there is no significant performance difference between Wukong and BitMat. TriAD also enables multi-threading and provides similar performance compared to Wukong for large (non-selective) queries. However, for small (selective) queries, Wukong is still at least an order-of-magnitude faster than TriAD due to the fast graph exploration, even without the optimizations aiming at distributed environment.

[7]LUBM-2560 is used due to limited main memory of a single machine, where the average (geometric mean) latency of Wukong on 6 machines is 7.5 msec.

We further compare Wukong with distributed systems with multi-threading enabled using LUBM-10240 in Table 3. For small queries (L4, L5 and L6), Wukong outperforms TriAD by up to 70.6X (from 8.4X) mainly due to the in-place execution with one-sided RDMA READ. For large queries (L1, L2, L3 and L7), Wukong still outperforms TriAD by up to 21.7X (from 4.1X), thanks to the fast graph exploration with indexing vertex and full-history pruning. The join-ahead pruning with summary graph (SG) improves the performance of TriAD, especially for L1 and L6, while Wukong still outperforms the average (geometric mean) latency of TriAD-SG by 9.0X (ranging from 2.8X to 26.6X). Compared to Trinity.RDF, which also uses graph-exploration strategy, the improvement of Wukong is at least one order of magnitude (from 10.1X to 78.0X), thanks to the full-history pruning that avoids redundant computation and communication as well as the time-consuming final join. Note that the result of Trinity.RDF is evaluated on a cluster with similar interconnects and twice the number of machines. SHARD is several orders of magnitude slower than other systems since it randomly partitions the RDF data and employs Hadoop as a communication layer for handling queries.

Table 4: The query latency (msec) of Wukong on evolving LUBM with 1 million triples/second ingestion rate.

LUBM-10240	L1	L2	L3	L4	L5	L6	L7
Wukong	587	87	222	0.43	0.18	0.95	516
Overhead (%)	12.0	10.3	8.6	4.7	5.6	6.3	10.1

Evolving RDF Graphs: To investigate the performance of Wukong on a continually growing graph, we ingest triples to the LUBM-10240 with the rate of 1 million triples per second on our 6-node cluster, while simultaneously handling queries. Currently, Wukong adopts a queries-friendly design, which minimizes the impact on query processing. The main overhead is from the versioning read. As shown in Table 4, the performance overhead of latency is only about 10.3% and 5.5% for large (L1, L2, L3 and L7) and small (L4, L5 and L6) queries respectively, depending on the number of data accessing.

7.3 Factor Analysis of Improvement

To study the impact of each design decision and how they affect the query performance, we iteratively enable each optimization and collect the query latency using the LUBM-10240 dataset, as shown in Table 5:

Table 5: The contribution of optimizations to query latency (msec) of Wukong. Optimizations are cumulative.

LUBM 10240	BASE	+RDMA	+FHP	+IDX	+PBS	+DYN
L1	9,766	9,705	888	853	814	516
L2	2,272	2,161	1,559	84	79	78
L3	421	404	404	205	203	203
L4	1.49	0.79	0.78	0.78	0.56	0.41
L5	1.00	0.39	0.39	0.39	0.31	0.17
L6	3.84	1.40	1.37	1.37	1.17	0.89
L7	2,176	2,041	657	494	466	464
Geo. M	102.3	69.1	39.6	22.6	19.9	15.7

- **BASE**: leverages graph-exploration strategy with one-step pruning. The communication adopts message passing over TCP/IP.
- **+RDMA**: uses one-sided RDMA operations to improve the communication.
- **+FHP**: enables full-history pruning (§5.1 and 5.2).
- **+IDX**: adds two types of index vertex (§4.1) and differentiated graph partitioning (§4.2).
- **+PBS**: leverages predicate-based finer-grained vertex decomposition (§4.3).
- **+DYN**: supports in-place execution and dynamically switches between data migration and execution distribution (§5.3).

Overall, all optimizations (**+DYN**) improves the average (geometric mean) latency by 6.5X over the basic version (**BASE**). The basic version already outperforms TriAD for small queries by leveraging graph exploration, while having inferior performance for large queries due to the overhead of the (expensive) final join operations. Note that Wukong can detect the empty final result of L3 in early steps and thus avoid the final join.

Leveraging RDMA for communication (**+RDMA**) improves the baseline performance slightly (ranging from 1% to 7%) for large queries and about twice (ranging from 1.9X to 2.7X) for small queries, depending on the proportion of communication cost. By skipping the costly final join, enabling full-history pruning (**+FHP**) notably accelerates the non-selective queries. The index vertex with differentiated partitioning (**+IDX**) can improve the parallelism and reduce network traffic for large queries launching from a set of entities (subject/object) with a certain predicate or type, especially for L2. L2 collects a large number of entities (i.e., Courses) on each machine, which can be avoided by decentralizing index vertex. Using predicate-based graph store (**+PBS**) further notably reduces the latency of small queries (ranging from 1.2X to 1.4X), due to finer-grained vertex decomposition by predicates. Finally, the *in-place* execution can bypass remote CPU and OS and avoid the overhead of task scheduling by leveraging one-sided RDMA READ to fetch remote data. Therefore, the optimization (**+DYN**) improves the performance by up to 1.8X.

Table 6: A comparison of query latency (msec) with different execution modes.

LUBM 10240	L1	L2	L3	L4	L5	L6	L7
In-place	21,859	80	204	0.42	0.17	2.43	12,068
Fork-join	813	79	203	0.63	0.47	1.27	466
Dynamic	516	78	203	0.41	0.17	0.89	464

To further study the benefit of dynamic execution mode switching in each step, we configure Wukong with a fixed mechanism (i.e. *in-place* or *fork-join*). As shown in Table 6, in-place mode is beneficial for L4 and L5, while fork-join execution is beneficial for L7. In addition, L2 and L3 are not sensitive to the choice of execution modes. L1 and L6 are relatively special, in which different steps require different modes for achieving optimal performance. Wukong can always choose the right mode in runtime and thus outperform in-place and fork-join mode alone by up to 42.3X and 2.8X. Note that the poor performance of L1 and L7 with in-place mode is caused by massive small-sized RDMA READs.

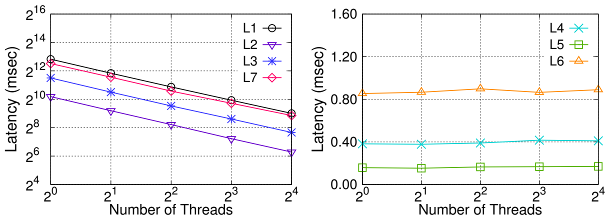


Fig. 14: The latency of queries in group (I) and (II) with the increase of threads on LUBM-10240.

7.4 Scalability

We evaluate the scalability of Wukong in three aspects by scaling the number of threads, the number of machines, and the size of dataset accordingly. We categorize seven queries on LUBM dataset into two groups according to the sizes of intermediate and final results as done in prior work [54]. Group (I): L1, L2, L3, and L7; the results of such queries increase with the growing of dataset. Group (II): L4, L5, and L6; such queries are quite selective and produce fixed-size results regardless of the data size.

Scale-up: We first study the performance impact of multi-threading on LUBM-10240 using fixed 6 servers. Figure 14 shows the latency of queries on a logarithmic scale with the logarithmic increase of threads. For group (I), the speedup of Wukong ranges from 9.9X to 14.3X with the increase of threads from 1 to 16. For group (II), since the queries just involve a small subgraph and are not CPU-intensive, Wukong always adopts a single thread for the query and provides a stable performance.

Scale-out: We also evaluated the scalability of Wukong with respect to the number of servers. Note that we omit the evaluation on a single server as LUBM-10240 (amounting to 230GB in raw NT format) cannot fit into memory. Figure 15(a) shows a linear speedup of

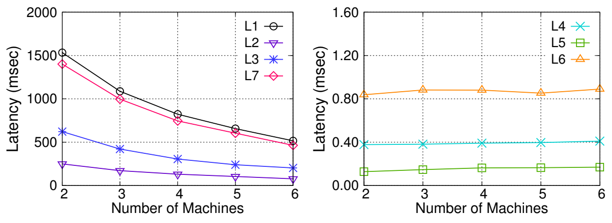


Fig. 15: The latency of queries in group (I) and (II) with the increase of machines on LUBM-10240.

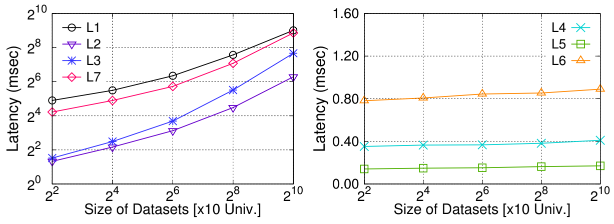


Fig. 16: The latency of queries in group (I) and (II) with the increase of LUBM datasets (40-10240).

Wukong for group (I) ranging from 2.46X to 3.54X, with the increase of servers from 2 to 6. It implies Wukong can efficiently utilize the parallelism of a distributed system by leveraging fork-join execution mode. For group (II), since the intermediate and final results are relatively small and fixed-size, using more machines does not improve the performance as expected, but the performance is still stable by using in-place execution to restrict the network overhead.

Data size: We further evaluated Wukong with the increase of dataset size from LUBM-40 to LUBM-10240 while keeping the number of threads and servers fixed. As shown in Figure 16, for group (I), Wukong scales quite well with the growing of dataset, due to efficiently passing full history and the elimination of the final join. For group (II), Wukong can achieve stable performance regardless of the increasing dataset size, due to the in-place execution with one-sided RDMA READ.

Wukong is a good practicer of the COST metric [29], which pursues scalable parallelism for large queries and efficient use of resources for small queries.

7.5 Throughput of Mixed Workloads

Unlike prior RDF stores [54, 21] that are only designed to handle one query at a time, Wukong is also designed to provide high throughput such that it can handle hundreds of thousands of concurrent queries per second. Therefore, we build emulated clients and a mixed workload to study the behavior of RDF stores serving concurrent queries.

For Wukong, each server runs up to 4 emulated clients on dedicated cores. All clients will send as many queries as possible periodically until the throughput saturated. For TriAD[8], a single client will send queries one by one

[8]We are not aware of open-sourced RDF systems supporting concurrent query processing. On the other hand, existing graph databases or graph-analytics systems have even worse performance compared to TriAD due to the lack of RDF and SPARQL supporting.

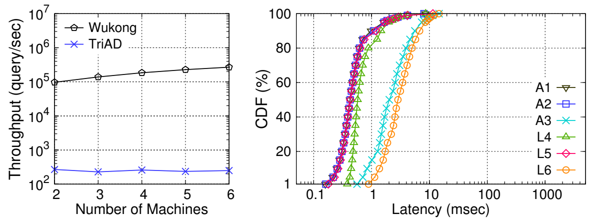


Fig. 17: (a) The throughput of a mixture of queries with the increase of machines, and (b) the CDF of latency for 6 classes of queries on 6 machines.

since it only can handle one query at a time.

We first use a mixture workload consisting of 6 classes of queries[9], all of which disable multi-threading. The query in each class has a similar behavior except that the start point is randomly selected from the same type of vertices (e.g., Univ0, Univ1, etc.). The distribution of query classes follows the reciprocal of their average latency. As shown in Figure 17, Wukong achieves a peak throughput of 269K queries/second on 6 machines (97K queries/second on 2 machines), which is at least two orders of magnitude higher than TriAD (from 278X to 740X). Under the peak throughput, the geometric mean of 50th (median) and 99th percentile latency is just 0.80 and 5.90 milliseconds respectively.

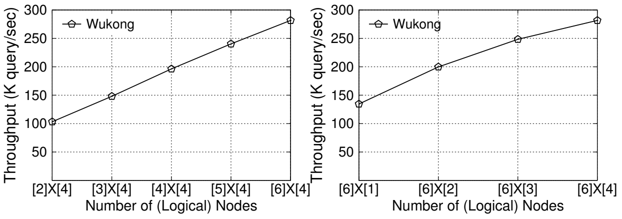


Fig. 18: The throughput of a mixture of queries with the increase of logical nodes. The tick labels of x-axis are the configuration, and the symbol of `[m]X[n]` corresponds with `#machines` and `#nodes/machine`.

Scalability with logical nodes: To overcome the restriction of cluster size, we emulate a large cluster by scaling the logical nodes on each machine and evaluate the throughput of Wukong along with the increase of logical nodes. Each logical node has 4 worker threads and the interaction between logical nodes still uses one-sided RDMA operations even on the same machine. As shown in Figure 18, Wukong scales out to 24 nodes by both the number of machines and the number of nodes per machine; the throughput reaches 282K queries per second.

Multi-threading query: To further study the impact of enabling multi-threading (MT) for time-consuming queries. We dedicate a client to continually send MT

[9]The templates of 6 classes of queries are based on group (II) queries (L4, L5, and L6) and three additional queries (A1, A2, and A3) from the official LUBM website (#1, #3, and #5).

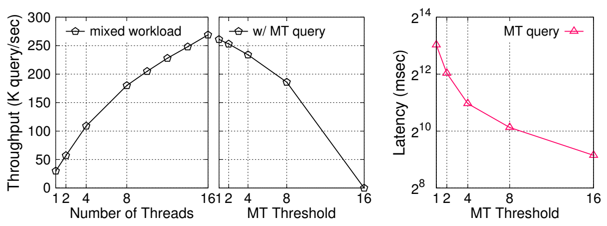


Fig. 19: (a) The throughput of a mixture of queries with the increase of threads, (b) the throughput with multi-threaded (MT) queries under various MT thresholds, and (c) the average latency of multi-threaded (MT) queries.

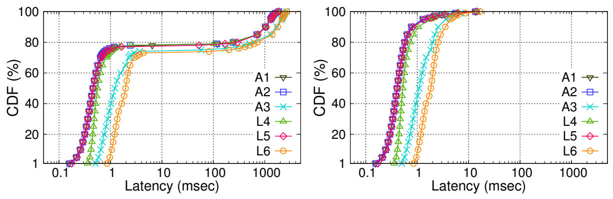


Fig. 20: The CDF of latency for 6 classes of queries on 6 machines (a) w/o and (b) w/ worker-obliger mechanism. Each server uses fixed 8 threads (threshold=4).

queries (i.e., L1) and configure Wukong with different MT thresholds. As shown in Figure 19(b) and (c), with the increase of the MT threshold, both the throughput of Wukong and the time of interference (the latency of MT query) will degrade. For example, under the threshold 8, Wukong can still perform 186K queries/second and the average latency of MT query is about 1,118 msec.

Worker-obliger mechanism: The MT query will also influence the latency of other small queries in the waiting queues. Figure 20(a) show the CDF graph of latency for 6 classes of non-MT queries. The 80th percentile latency increases at least two orders of magnitude and the 99th percentile latency reaches several thousands of msec. With the worker-obliger mechanism, as shown in Figure 20(b), Wukong can notably reduce the query latency while preserving the throughput.

LUBM 2560	**MEM** (GB)
Wukong	18.5
BASE	12.3
TriAD	30.7
RDF-3X	24.2
BitMat	27.4

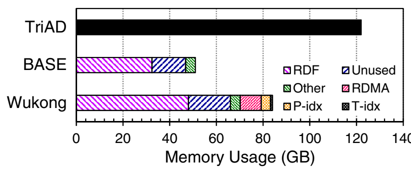


Fig. 21: A comparison of memory usage and breakdown on various systems for (a) LUBM-2560 and (b) LUBM-10240. The storage size is 6.2GB and 25GB respectively.

7.6 Memory Consumption

Readers might be interested in how the memory consumption of Wukong compares to other state-of-the-art systems. Triple stores, including TriAD, RDF-3X, and BitMat, rely on redundant *six* primary SPO permutation indexes [49] to accelerate querying, which, however, lead to high memory pressure. In contrast, managing RDF data in native graph form is much space-efficient, which only doubles the triples in RDF for subjects and objects. Figure 21(a) compares the memory usage of various systems for LUBM-2560 on a single machine. All triple stores consume much more memory compared to Wukong, especially for its basic version (i.e., BASE).

Figure 21(b) further shows a breakdown of memory usage in Wukong for LUBM-10240 on the 6-node cluster. Compared to the base version, Wukong adds about 3.9GB and 0.9GB memory for predicate index (P-idx) and type index (T-idx), as well as additional 15.5GB memory for RDF to support predicate-based store. Furthermore, 9.0GB memory (1.5GB per machine) is reserved for one-sided RDMA operations. Note that the underlying key/value store of Wukong is a hashtable with less than 75% occupancy, because Wukong is currently not well tuned for high space-efficiency.

7.7 Other Datasets

We further study the performance of Wukong and TriAD over more other synthetic and real-life datasets. Note that we do not provide the performance of TriAD-SG because the hand-tuned parameter of summary graph is not known and it only improves performance in few cases.

Table 7: The latency (msec) of queries on WSDTS

WSDTS	**L1-L5** (Geo. M)	**S1-S7** (Geo. M)	**F1-F5** (Geo. M)	**C1-C3** (Geo. M)
TriAD	4.5	5.3	17.5	36.6
Wukong	1.0	0.9	3.6	10.3

WSDTS: We first compare the performance of TriAD and Wukong over WSDTS dataset using 20 diverse queries, which are classified into linear (L), star (S), snowflake (F) and complex (C). Table 7 shows the geometric mean of latency for various query classes. Wukong always outperforms TriAD by up to 58.2X (from 1.6X). For L1, L3, S1, S7 and F5, Wukong is at least one order of magnitude faster than TriAD since the queries are quite selective and appropriate for graph exploration. For only two queries, F1 and C3, the improvement of Wukong is less than 2.0X.

Table 8: The latency (msec) of queries on DBPSB

DBPSB	**D1**	**D2**	**D3**	**D4**	**D5**	**Geo. M**
TriAD	4.93	4.10	5.56	7.68	3.51	4.97
Wukong	1.75	0.48	0.41	3.70	1.14	1.16

DBPSB: Table 8 shows the performance of five representative queries on DBPSB, which is a relative small real-life dataset, but has quite more predicates. Wukong outperforms TriAD by at least 2X (up to 13.6X), and the improvement of geometric mean reaches 4.3X. For D2 and D3, the speedup reaches 8.6X and 13.6X respectively since the queries are relatively selective.

YAGO2: Table 9 compares the performance of TriAD and Wukong on a large real-life dataset YAGO2. For the

Table 9: The latency (msec) of queries on YAGO2

YAGO2	Y1	Y2	Y3	Y4	Geo. M
TriAD	1.13	2.14	68,841	6,193	179
Wukong	0.12	0.17	38,571	3,501	41

simple queries, Y1 and Y2, Wukong is one order of magnitude faster than TriAD due to fast in-place execution. For the complex queries, Y3 and Y4, Wukong can still notably outperforms TriAD by about 1.8X due to full-history pruning and RDMA-friendly task queues.

8 Related Work

RDF query over triple and relational store: There have been a large number of triple-based RDF stores that use relational approaches to storing and indexing RDF data [33, 34, 8, 49, 42, 11]. Since join is expensive and a key step for query processing in such triple stores, they perform various query optimizations including heuristic optimizations [33], join-ordering exploration [33], join-ahead pruning [34], and query caching [39]. Specially, TriAD [21] is a recent distributed in-memory RDF engine that leverages join-ahead pruning and graph summarization with asynchronous message passing for parallelization. SHAPE [27] is a distributed engine upon RDF-3X by statically replicating and prefetching data. As shown in prior work [54], graph exploration avoids many redundant immediate results generated during expensive join operations and thus typically delivers better performance. A recent study, SQLGraph [45], leverages a relational store to store RDF data but processes RDF queries as a graph store. Yet, it focuses on query rewriting and schema refinement to support ACID-style transactions and thus has different objectives from Wukong.

RDF query over graph store: There is an increasing interest in using native graph model to store and query RDF data [9, 53, 58, 52, 54]. BitMat [9], gStore [58] and TripleBit [53] are centralized graph stores with sophisticated indexes to improve query performance. Sedge [52] is a distributed SPARQL query engine based on a simple Pregel implementation, which tries to minimize the inter-machine communication by group-based communication. The most related work is Trinity.RDF [54], a distributed in-memory RDF store that leverages graph exploration to process queries. Wukong's design centers around the usage of fast interconnect with RDMA features to allow fast graph exploration. Wukong also introduces novel graph-based indexes as well as differentiated graph partitioning and query processing to improve the overall system performance.

RDF query over MapReduce: Several distributed RDF systems are built atop existing frameworks like MapReduce [38, 37, 40, 43], e.g., H_2RDF [38, 37] and SHARD [40]. PigSPARQL [43] maps SPARQL operations into PigLatin [35] queries, which in turn is translated into MapReduce programs. However, due to the lack of efficient iterative computation support, MapReduce-based computation is usually sub-optimal for SPARQL execution, as shown in prior work [21, 54].

Graph databases and query systems: Neo4j [2] and HyperGraphDB [24] focus on supporting online transaction processing (OLTP) on graph data; however they are not distributed and cannot support web-scale graphs partitioned over multiple machines. Titan [4] instead supports distributed graph traversals over multiple machines, which, however, does not support SPARQL queries. Facebook's TAO [12] provides a simple API and data model to store and query geographically distributed data. Unicorn [15] further leverages TAO as the storage layer to support searching over the social data. To our knowledge, none of the above systems exploit RDMA as well as the optimization techniques in Wukong to boost query latency and throughput.

RDMA-centric stores: The low latency and high throughput of RDMA-based networking stimulate much work on RDMA-centric key/value stores [30, 25], OLTP platforms [48, 17, 14] and graph analytics engines [50, 23]. Specifically, GraM [50] is an efficient and scalable graph analytics engine that leverages multicore and RDMA to provide fast batch-oriented graph analytics. However, handling SPARQL queries is significantly different from graph analytics and thus Wukong can hardly benefit from the design of GraM. Further, Wukong is designed to handle highly concurrent queries while GraM is designed to handle one graph-analytics task at a time. Recently, Kalia et al. [26] provide several of RDMA design space for system designers.

9 Conclusion

This paper describes Wukong, a distributed in-memory RDF store that leverages RDMA-based graph exploration to support fast and concurrent SPARQL queries. Wukong significantly outperforms state-of-the-art systems and can process a mixture of small and large queries at 269K queries/second on a 6-node RDMA-capable cluster. Currently, we only consider the SPARQL query over timeless RDF datasets; our future work may extend Wukong to support RDF stream processing (RSP)[10].

10 Acknowledgments

We sincerely thank our shepherd Dushyanth Narayanan and the anonymous reviewers for their insightful suggestions, as well as Yunhao Zhang for sharing his experience to support evolving RDF graphs. This work is supported in part by the National Key Research & Development Program (No. 2016YFB1000500), the National Natural Science Foundation of China (No. 61402284, 61572314) and the Zhangjiang Hi-Tech program (No. 201501-YP-B108-012).

[10]https://www.w3.org/community/rsp/

References

[1] DBpedias SPARQL Benchmark. http://aksw.org/Projects/DBPSB.

[2] Neo4j Graph Database. http://neo4j.org/.

[3] SWAT Projects - the Lehigh University Benchmark (LUBM). http://swat.cse.lehigh.edu/projects/lubm/.

[4] Titan: Distributed Graph Database. http://titan.thinkaurelius.com/.

[5] Waterloo SPARQL Diversity Test Suite (WSDTS). https://cs.uwaterloo.ca/~galuc/wsdts/.

[6] YAGO: A High-Quality Knowledge Base. http://www.mpi-inf.mpg.de/departments/databases-and-information-systems/research/yago-naga/yago.

[7] Bio2RDF: Linked Data for the Life Science. http://bio2rdf.org/, 2014.

[8] ABADI, D. J., MARCUS, A., MADDEN, S. R., AND HOLLENBACH, K. Sw-store: a vertically partitioned dbms for semantic web data management. *The VLDB JournalThe International Journal on Very Large Data Bases 18*, 2 (2009), 385–406.

[9] ATRE, M., CHAOJI, V., ZAKI, M. J., AND HENDLER, J. A. Matrix "bit" loaded: A scalable lightweight join query processor for rdf data. In *Proceedings of the 19th International Conference on World Wide Web* (New York, NY, USA, 2010), WWW '10, ACM, pp. 41–50.

[10] BLUMOFE, R. D., AND LEISERSON, C. E. Scheduling multithreaded computations by work stealing. *J. ACM 46*, 5 (Sept. 1999), 720–748.

[11] BORNEA, M. A., DOLBY, J., KEMENTSIETSIDIS, A., SRINIVAS, K., DANTRESSANGLE, P., UDREA, O., AND BHATTACHARJEE, B. Building an efficient rdf store over a relational database. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data* (New York, NY, USA, 2013), SIGMOD '13, ACM, pp. 121–132.

[12] BRONSON, N., AMSDEN, Z., CABRERA, G., CHAKKA, P., DIMOV, P., DING, H., FERRIS, J., GIARDULLO, A., KULKARNI, S., LI, H., ET AL. Tao: Facebooks distributed data store for the social graph. In *Presented as part of the 2013 USENIX Annual Technical Conference (USENIX ATC 13)* (2013), pp. 49–60.

[13] CHEN, R., SHI, J., CHEN, Y., AND CHEN, H. Powerlyra: Differentiated graph computation and partitioning on skewed graphs. In *Proceedings of the Tenth European Conference on Computer Systems* (New York, NY, USA, 2015), EuroSys '15, ACM, pp. 1:1–1:15.

[14] CHEN, Y., WEI, X., SHI, J., CHEN, R., AND CHEN, H. Fast and general distributed transactions using rdma and htm. In *Proceedings of the Eleventh European Conference on Computer Systems* (2016), ACM, p. 26.

[15] CURTISS, M., BECKER, I., BOSMAN, T., DOROSHENKO, S., GRIJINCU, L., JACKSON, T., KUNNATUR, S., LASSEN, S., PRONIN, P., SANKAR, S., SHEN, G., WOSS, G., YANG, C., AND ZHANG, N. Unicorn: A system for searching the social graph. *Proc. VLDB Endow. 6*, 11 (Aug. 2013), 1150–1161.

[16] DRAGOJEVIĆ, A., NARAYANAN, D., HODSON, O., AND CASTRO, M. FaRM: Fast remote memory. In *Proceedings of the 11th USENIX Conference on Networked Systems Design and Implementation* (2014), NSDI'14, USENIX Association, pp. 401–414.

[17] DRAGOJEVIĆ, A., NARAYANAN, D., NIGHTINGALE, E. B., RENZELMANN, M., SHAMIS, A., BADAM, A., AND CASTRO, M. No compromises: Distributed transactions with consistency, availability, and performance. In *Proceedings of the 25th Symposium on Operating Systems Principles* (New York, NY, USA, 2015), SOSP'15, ACM, pp. 54–70.

[18] GONZALEZ, J., LOW, Y., GU, H., BICKSON, D., AND GUESTRIN, C. PowerGraph: Distributed graph-parallel computation on natural graphs. In *OSDI* (2012).

[19] GONZALEZ, J. E., XIN, R. S., DAVE, A., CRANKSHAW, D., FRANKLIN, M. J., AND STOICA, I. Graphx: Graph processing in a distributed dataflow framework. In *OSDI* (2014).

[20] GOOGLE INC. Introducing the knowledge graph: things, not strings. https://googleblog.blogspot.co.uk/2012/05/introducing-knowledge-graph-things-not.html, 2012.

[21] GURAJADA, S., SEUFERT, S., MILIARAKI, I., AND THEOBALD, M. Triad: A distributed shared-nothing rdf engine based on asynchronous message passing. In *Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data* (New York, NY, USA, 2014), SIGMOD '14, ACM, pp. 289–300.

[22] HOFFART, J., SUCHANEK, F. M., BERBERICH, K., LEWIS-KELHAM, E., DE MELO, G., AND WEIKUM, G. Yago2: Exploring and querying world knowledge in time, space, context, and many languages. In *Proceedings of the 20th International Conference Companion on World Wide Web* (New York, NY, USA, 2011), WWW'11, ACM, pp. 229–232.

[23] HONG, S., DEPNER, S., MANHARDT, T., VAN DER LUGT, J., VERSTRAATEN, M., AND CHAFI, H. Pgx.d: A fast distributed graph processing engine. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis* (New York, NY, USA, 2015), SC '15, ACM, pp. 58:1–58:12.

[24] IORDANOV, B. Hypergraphdb: a generalized graph database. In *Web-Age information management*. Springer, 2010, pp. 25–36.

[25] KALIA, A., KAMINSKY, M., AND ANDERSEN, D. G. Using rdma efficiently for key-value services. In *Proceedings of the 2014 ACM Conference on SIGCOMM* (2014), SIGCOMM'14, ACM, pp. 295–306.

[26] KALIA, A., KAMINSKY, M., AND ANDERSEN, D. G. Design guidelines for high performance rdma systems. In *Proceedings of the 2016 USENIX Conference on Usenix Annual Technical Conference* (2016), USENIX ATC'16.

[27] LEE, K., AND LIU, L. Scaling queries over big rdf graphs with semantic hash partitioning. *Proc. VLDB Endow. 6*, 14 (Sept. 2013), 1894–1905.

[28] MALEWICZ, G., AUSTERN, M. H., BIK, A. J., DEHNERT, J. C., HORN, I., LEISER, N., AND CZAJKOWSKI, G. Pregel: a system for large-scale graph processing. In *SIGMOD* (2010), pp. 135–146.

[29] MCSHERRY, F., ISARD, M., AND MURRAY, D. Scalability! But at what COST? In *HotOS '15* (2015).

[30] MITCHELL, C., GENG, Y., AND LI, J. Using one-sided rdma reads to build a fast, cpu-efficient key-value store. In *USENIX Annual Technical Conference* (2013), pp. 103–114.

[31] MURRAY, D., MCSHERRY, F., ISAACS, R., ISARD, M., BARHAM, P., AND ABADI, M. Naiad: a timely dataflow system. In *SOSP* (2013).

[32] NATIONAL CENTER FOR BIOTECHNOLOGY INFORMATION. PubChemRDF. https://pubchem.ncbi.nlm.nih.gov/rdf/, 2014.

[33] NEUMANN, T., AND WEIKUM, G. Rdf-3x: A risc-style engine for rdf. *Proc. VLDB Endow. 1*, 1 (Aug. 2008), 647–659.

[34] NEUMANN, T., AND WEIKUM, G. Scalable join processing on very large rdf graphs. In *Proceedings of the 2009 ACM SIGMOD International Conference on Management of Data* (New York, NY, USA, 2009), SIGMOD '09, ACM, pp. 627–640.

[35] OLSTON, C., REED, B., SRIVASTAVA, U., KUMAR, R., AND TOMKINS, A. Pig latin: a not-so-foreign language for data processing. In *Proceedings of the 2008 ACM SIGMOD international conference on Management of data* (2008), ACM, pp. 1099–1110.

[36] OUSTERHOUT, K., WENDELL, P., ZAHARIA, M., AND STOICA, I. Sparrow: Distributed, low latency scheduling. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles* (New York, NY, USA, 2013), SOSP '13, ACM, pp. 69–84.

[37] PAPAILIOU, N., KONSTANTINOU, I., TSOUMAKOS, D., KARRAS, P., AND KOZIRIS, N. H2rdf+: High-performance distributed joins over large-scale rdf graphs. In *2013 IEEE International Conference on Big Data* (2013), IEEE BigData '13, IEEE, pp. 255–263.

[38] PAPAILIOU, N., KONSTANTINOU, I., TSOUMAKOS, D., AND KOZIRIS, N. H2rdf: Adaptive query processing on rdf data in the cloud. In *Proceedings of the 21st International Conference on World Wide Web* (New York, NY, USA, 2012), WWW '12 Companion, ACM, pp. 397–400.

[39] PAPAILIOU, N., TSOUMAKOS, D., KARRAS, P., AND KOZIRIS, N. Graph-aware, workload-adaptive sparql query caching. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data* (New York, NY, USA, 2015), SIGMOD '15, ACM, pp. 1777–1792.

[40] ROHLOFF, K., AND SCHANTZ, R. E. High-performance, massively scalable distributed systems using the mapreduce software framework: The shard triple-store. In *Programming Support Innovations for Emerging Distributed Applications* (New York, NY, USA, 2010), PSI EtA '10, ACM, pp. 4:1–4:5.

[41] ROY, A., BINDSCHAEDLER, L., MALICEVIC, J., AND ZWAENEPOEL, W. Chaos: Scale-out graph processing from secondary storage. In *Proceedings of the 25th Symposium on Operating Systems Principles* (New York, NY, USA, 2015), SOSP '15, ACM, pp. 410–424.

[42] SAKR, S., AND AL-NAYMAT, G. Relational processing of rdf queries: A survey. *SIGMOD Rec. 38*, 4 (June 2010), 23–28.

[43] SCHÄTZLE, A., PRZYJACIEL-ZABLOCKI, M., AND LAUSEN, G. Pigsparql: Mapping sparql to pig latin. In *Proceedings of the International Workshop on Semantic Web Information Management* (2011), ACM, p. 4.

[44] SHAO, B., WANG, H., AND LI, Y. Trinity: A distributed graph engine on a memory cloud. In *SIGMOD* (2013).

[45] SUN, W., FOKOUE, A., SRINIVAS, K., KEMENTSIETSIDIS, A., HU, G., AND XIE, G. Sqlgraph: An efficient relational-based property graph store. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data* (New York, NY, USA, 2015), SIGMOD '15, ACM, pp. 1887–1901.

[46] THOMSON, A., DIAMOND, T., WENG, S.-C., REN, K., SHAO, P., AND ABADI, D. J. Calvin: Fast distributed transactions for partitioned database systems. In *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data* (2012), SIGMOD'12, ACM, pp. 1–12.

[47] TU, S., ZHENG, W., KOHLER, E., LISKOV, B., AND MADDEN, S. Speedy transactions in multicore in-memory databases. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles* (2013), SOSP'13, ACM, pp. 18–32.

[48] WEI, X., SHI, J., CHEN, Y., CHEN, R., AND CHEN, H. Fast in-memory transaction processing using rdma and htm. In *Proceedings of the 25th Symposium on Operating Systems Principles* (New York, NY, USA, 2015), SOSP '15, ACM, pp. 87–104.

[49] WEISS, C., KARRAS, P., AND BERNSTEIN, A. Hexastore: Sextuple indexing for semantic web data management. *Proc. VLDB Endow. 1*, 1 (Aug. 2008), 1008–1019.

[50] WU, M., YANG, F., XUE, J., XIAO, W., MIAO, Y., WEI, L., LIN, H., DAI, Y., AND ZHOU, L. Gram: Scaling graph computation to the trillions. In *Proceedings of the Sixth ACM Symposium on Cloud Computing* (New York, NY, USA, 2015), SoCC '15, ACM, pp. 408–421.

[51] WU, W., LI, H., WANG, H., AND ZHU, K. Q. Probase: A probabilistic taxonomy for text understanding. In *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data* (2012), ACM, pp. 481–492.

[52] YANG, S., YAN, X., ZONG, B., AND KHAN, A. Towards effective partition management for large graphs. In *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data* (New York, NY, USA, 2012), SIGMOD '12, ACM, pp. 517–528.

[53] YUAN, P., LIU, P., WU, B., JIN, H., ZHANG, W., AND LIU, L. Triplebit: A fast and compact system for large scale rdf data. *Proc. VLDB Endow. 6*, 7 (May 2013), 517–528.

[54] ZENG, K., YANG, J., WANG, H., SHAO, B., AND WANG, Z. A distributed graph engine for web scale rdf data. In *Proceedings of the 39th international conference on Very Large Data Bases* (2013), PVLDB'13, VLDB Endowment, pp. 265–276.

[55] ZHANG, M., WU, Y., CHEN, K., QIAN, X., LI, X., AND ZHENG, W. Exploring the hidden dimension in graph processing. In *OSDI* (2016).

[56] ZHU, X., CHEN, W., ZHENG, W., AND XIAOSONG, M. Gemini: A computation-centric distributed graph processing system. In *OSDI* (2016).

[57] ZHU, Y., ERAN, H., FIRESTONE, D., GUO, C., LIPSHTEYN, M., LIRON, Y., PADHYE, J., RAINDEL, S., YAHIA, M. H., AND ZHANG, M. Congestion control for large-scale rdma deployments. In *Proceedings of the 2015 ACM Conference on Special Interest Group on Data Communication* (New York, NY, USA, 2015), SIGCOMM '15, ACM, pp. 523–536.

[58] ZOU, L., MO, J., CHEN, L., ÖZSU, M. T., AND ZHAO, D. gstore: Answering sparql queries via subgraph matching. *Proc. VLDB Endow. 4*, 8 (May 2011), 482–493.

RE^X: A Development Platform and Online Learning Approach for Runtime Emergent Software Systems

Barry Porter[†], Matthew Grieves[†], Roberto Rodrigues Filho[†] and David Leslie[‡]
[†]*School of Computing and Communications;* [‡]*Department of Mathematics and Statistics*
Lancaster University, UK

Abstract: Conventional approaches to self-adaptive software architectures require human experts to specify models, policies and processes by which software can adapt to its environment. We present RE^X, a complete platform and online learning approach for *runtime emergent software systems*, in which all decisions about the *assembly* and *adaptation* of software are machine-derived. RE^X is built with three major, integrated layers: (i) a novel component-based programming language called Dana, enabling discovered assembly of systems and very low cost adaptation of those systems for dynamic re-assembly; (ii) a perception, assembly and learning framework (PAL) built on Dana, which abstracts emergent software into configurations and perception streams; and (iii) an online learning implementation based on a linear bandit model, which helps solve the search space explosion problem inherent in runtime emergent software. Using an emergent web server as a case study, we show how software can be autonomously self-assembled from discovered parts, and continually optimized over time (by using alternative parts) as it is subjected to different deployment conditions. Our system begins with no knowledge that it is specifically assembling a web server, nor with knowledge of the deployment conditions that may occur at runtime.

1 Introduction

Modern software systems are increasingly complex, and are deployed into increasingly dynamic environments. The result is systems comprising millions of lines of code that are designed, analyzed and maintained by large teams of software developers at significant cost. It is broadly acknowledged that this level of complexity is unsustainable using current practice [15]. In recent years this has driven research in autonomic, self-adaptive and self-organizing software systems [26, 31, 14], aiming to move selected responsibility for system management into the software itself. While showing promise, work to date retains a very high degree of human involvement – either in creating models to describe systems and their adaptation modes [10, 13], policies to control adaptation at runtime [20], or designing and running courses of offline training with available historical data [12]. These are *human-led* approaches to the above complexity problem, designed to fit well with current software development practice.

We push these concepts to their limits with a novel *machine-led* approach, in which a software system autonomously *emerges* from a pool of available building blocks that are provided to it. We demonstrate the first such example of a software system able to rapidly self-assemble into an optimal form, at runtime, using online learning. This is done with no models or architecture specifications, and no policies for adaptation. Instead, the live system learns by assembling itself from needed behaviors and continually perceiving their effectiveness, such as response time or compression ratio, in the environments to which the system is subjected. The building blocks of our approach are based on *micro-variation*: different implementations of small software components such as memory caches with different cache replacement strategies or stream handlers that do or do not use caching. As we use relatively small components, this kind of implementation variant is easy to create. By autonomously assembling systems from these micro-variations, and their various combinations, we then see emergent designs to suit the conditions observed at runtime.

Implemented as a development platform called RE^X, we present three major integrated contributions, each a key part of the solution to *emergent computer software*:

- **An implementation platform**: We present the key features of *Dana*, a programming language with which to create small software components that can be assembled into emergent software systems. Dana offers a uniform way to express systems in these terms, and near-zero-cost runtime adaptation.
- **A perception, assembly and learning framework**: We present the details of *PAL*, a framework built with Dana that controls the dynamic discovery and assembly of emergent software, perceives the effectiveness and deployment conditions of that software (such as input patterns and system load characteristics), and feeds perception data to an online learning module.
- **An online learning approach**: We present an application of statistical linear bandits, using Thompson sampling, as an effective online learning algorithm that helps to solve the search space explosion inherent in emergent software. This is done by sharing beliefs about individual components across the configurations in which they can appear.

Using a prototype emergent web server as an example, we show how a system can be autonomously assembled from discovered parts, and how that system can subsequently be optimized to its task by seamlessly re-assembling it from alternative parts. We evaluate our approach by subjecting our web server to various usage patterns, demonstrating how different designs rapidly emerge over time as conditions change. This emergence occurs through online learning, based on perception streams that indicate the internal well-being of the software and the external conditions to which it is currently being subjected. We also show how a simple classifier adds "memory" to the system, avoiding re-learning of environmental change. In our current implementation this classifier is manually defined; further automation here is a topic of future work.

Our work paves the way to: (i) significantly reducing human involvement in software development, thereby reducing the scale of modern development processes; and (ii) creating systems that are far more responsive to the actual conditions that they encounter at runtime, therefore offering higher performance in those conditions.

The remainder of this paper is structured as follows. In Sec. 2 we discuss our approach in detail, presenting the above three contributions and how they integrate into a complete platform for emergent computer software. In Sec. 3 we then evaluate the system in terms of its ability to continuously (and rapidly) assemble optimal software compositions as external stimulus changes. In Sec. 4 we discuss related work and we conclude the paper in Sec. 5.

2 Approach

Our approach has three major contributions that build on each other to provide an integrated solution for emergent software. An overview is shown in Fig. 1.

At the bottom layer is our implementation platform (Dana) for creating software components that can be assembled / re-assembled in various ways into different systems. This layer provides an API to control the loading, unloading and interconnection of these components. The upper two layers then contain our perception, assembly and learning (PAL) framework. Specifically, at the middle layer are our assembly and perception modules, responsible for assembling entire systems from available components and perceiving the state of those systems. Together these two modules offer an API to control the way in which the system is currently assembled, and to view the perception streams that the system is emitting. The top layer contains our learning module, which uses the assembly and perception API. This module learns correlations between particular assemblies of behavior and the way that the system perceives its own well-being, under different external stimuli of input patterns or deployment environment conditions such as CPU load.

Our overall approach uses dynamic *micro-variation* of

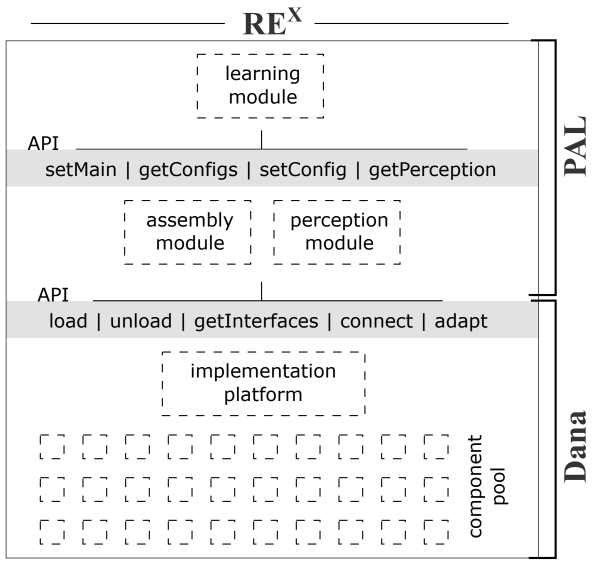


Figure 1 – Overview of our approach.

behavior: the ability to continually discover and configure components in and out of a live system that perform the same overall task but do so in different ways. By experimenting with these variations, and their combinations, we see the ideal system emerging over time for the current usage pattern and deployment environment conditions[1].

In the following sections we present Dana; our PAL framework; and the online learning approach used in PAL.

2.1 Dana: An implementation platform for runtime adaptive micro-variation

A platform for emergent software must enable us to build small units of behavior, and to express the interrelationships between them, such that we can create micro-variations of these units that can be autonomously assembled, and seamlessly re-assembled, in a live system. To do this we started from the component-based software development paradigm [29], well-established in forging adaptive systems. We designed a programming language around this, in which all elements of a system (from abstract data types to GUI widgets) are runtime-replaceable components. Our language is called *Dana*[2] and is freely available [1], with a large standard library of components. It is currently used across a range of ongoing projects.

Dana is a multi-threaded imperative programming language, but one that frames these concepts in a component-based structural paradigm. In these terms, Dana has three novel features for our needs that we now present in detail, along with the API that Dana provides to higher layers of RE^X for assembling and perceiving emergent software.

[1]Throughout this paper we use the simplifying assumption that all possible assemblies of an emergent system are valid; in reality an automated unit testing system could potentially provide this validation before particular assemblies are made available for use in the live system.

[2]Dynamic Adaptive Nucleic Architectures: named for its highly dynamic systems of small components with linked internal sub-structures.

```
interface File {
  transfer char path[]
  transfer int pos, mode

  File(char path[], int mode)
  byte[] read(int numBytes)
  int write(byte data[])
  bool eof()
  void close()
}
```

```
component provides App requires File {
  int App:main(AppParam args[]) {
     File ifd = new File(args[0].str, File.READ)
     File ofd = new File(args[1].str, File.WRITE)
     while (!ifd.eof()) ofd.write(ifd.read(128))
     ofd.close()
     ifd.close()
     return 0
  }
}
```

Figure 2 – Example interface to open, read and write files (top); and a component that uses this interface to copy a file (bottom).

2.1.1 Fusing third-party system composition with first-party instantiation

Our first observation is that state-of-the-art realizations of the component-based paradigm are almost completely disjoint from object orientation. Specifically, component models enable third-party instantiation and (re)wiring whereby a so-called *meta-level* controls the composition of a software system by modelling that system as a graph of components (nodes) and wirings (edges). However, within this model they fundamentally lack support for first-party instantiation and reference passing – i.e., the ability to instantiate objects with their own private state and pass references to those objects as parameters. In our experience with existing runtime component models (such as OSGi [3], OpenCom [11] and Fractal [8]), this shortcoming makes it very hard to express many simple modern programming concepts. By contrast, Dana enables first-party instantiation within a paradigm in which a meta-level controls software composition as a graph of components and wirings that can be adapted at runtime.

An example Dana component is shown in the lower half of Fig. 2. This component *provides* an `App` interface and *requires* a `File` interface (defined in the top half of Fig. 2). The component instantiates the `File` interface twice with different parameters, yielding two `File` objects, and copies data from the first file to the second by reading and writing chunks of data. The full system is created by a meta-level which first loads the example component (C_A) into memory, queries its required interfaces, then loads a desired implementing component of `File` into memory and wires C_A's required interface to the respective provided interface of that component. The resulting system is illustrated in Fig. 3. At any point during program execution, the meta-level can choose to rewire the `File` required interface of C_A to point to a corresponding provided interface on a different implementing component, thereby adapting the system's behavior.

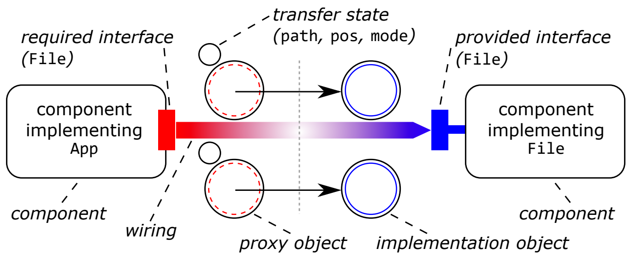


Figure 3 – Internal structure of components and objects. For each instantiated object, a proxy is created with an internal reference to the implementing object from the component of the connected provided interface. Required interfaces maintain a list of all (proxy) objects that were instantiated through them.

Formally, the runtime component model that enables the above is defined as follows. An interface i is a set of *function prototypes*, each comprising a function name, return type and parameter types; and a set of *transfer fields*, typed pieces of state that persist across alternate implementations of the interface during runtime adaptation.

A component c provides one or more such interfaces, where each such provided interface has an implementation scope i_{sc}. An i_{sc} has an implementation of every function of its provided interface (plus other internal functions) and $[0..n]$ global variables (each of which is private, i.e., not visible outside c). A provided interface (and its underlying i_{sc}) of a component must be *instantiated* before use; we refer to these instances as objects. A component c also requires zero or more interfaces, each of which must be connected to a compatible provided interface on another component to satisfy the dependency. These inter-component connections are referred to as wirings.

At an implementation level, a required interface can be thought of simply as a list of function pointers; when a required interface r is wired to a provided interface p, r's function pointers are updated to point at the corresponding functions in the component behind p. On top of this basic mechanism, Dana provides an abstraction of objects such that a required interface can be instantiated many times.

When the code inside a component instantiates one of its required interfaces r (using the language's `new` operator), this results in the instantiation firstly of a special *proxy object* and secondly an i_{sc} instance relative to the provided interface of the component to which r is wired. The i_{sc} has a fresh copy of any (private) global state fields, and of its interface's transfer fields. The proxy object has an internal link to the corresponding i_{sc}. A reference to the proxy object, initially held by the instantiator, can then be passed as a parameter to other functions as desired.

The use of proxy objects allows the implementing component(s) of those objects to be adapted, resulting in a change to the internal links within the proxy objects, without affecting any references to the proxy objects.

2.1.2 A protocol for seamless runtime adaptation

An emergent software system must be able to continually experiment online with different combinations of components without disrupting the system's primary task. To enable this we need a low-overhead runtime adaptation protocol that works with our component model.

We have therefore designed a protocol that uses transparent hot-swapping (similar to that proposed by Soules *et al.* [28]) for zero down-time; and that supports object reference persistence across implementation changes. In addition, it yields fast adaptation for both stateful and stateless components by using a modular protocol design in which different command orderings give different semantics for these two cases, as we explain below. This is much more lightweight than classic 'quiescence'-based approaches for coarse-grained component models, in which potentially large portions of a running system must be deactivated before adaptation proceeds [21, 32].

Algorithm 1 Adaptation protocol

```
 1: srcCom          ▷ Comp. to rewire a required interface of
 2: sinkCom         ▷ Comp. with provided interface to wire to
 3: intfName        ▷ Interface name being adapted
 4: pause(srcCom.intfName)
 5: $r_{objs}$ = getObjects(srcCom.intfName)
 6: rewire(srcCom.intfName, sinkCom)
 7: resume(srcCom.intfName)
 8: for $i = 0$ to $r_{objs}.arrayLength - 1$ do
 9:     if pauseObject($r_{objs[i]}$) then
10:         $a$ = adaptConstruct(sinkCom.intfName, $r_{objs[i]}$)
11:         $b$ = rewireObject($r_{objs[i]}$, $a$)
12:         resumeObject($r_{objs[i]}$)
13:         waitForObject($b$)
14:         adaptDestroy($b$)
15:     end if
16: end for
```

Pseudocode of our protocol is given in Alg. 1. All adaptation in Dana is performed by changing a component *srcCom*'s required interface *r* from its current wiring to instead be wired against a compatible provided interface of a different component *sinkCom*.

In simple terms, our two flavours of adaptation proceed as follows. For stateless objects, any new calls on those objects are immediately routed to their new implementation from *sinkCom*, while any existing calls active in the old implementation are (concurrently) allowed to finish.

For stateful objects, any existing calls active in their old implementations are allowed to finish while any new calls are temporarily held at the point of invocation, allowing existing calls to finish potential updates to transfer state fields. When all existing calls finish, all held calls, and any new calls, are then allowed to proceed and are routed to the objects' new implementations from *sinkCom*.

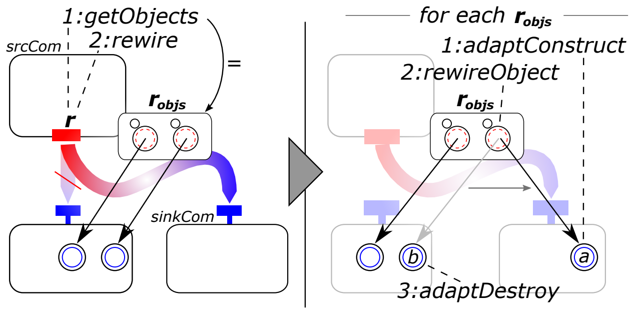


Figure 4 – Adaptation sequence overview. A selected required interface *r* is rewired, followed by each object in the set r_{objs}.

To achieve these effects, the operations used in our adaptation protocol are defined in detail as follows.

`pause` prevents new objects from being instantiated via *r*, and prevents any existing instances from being destroyed. Specifically, any Dana language instructions that attempt to instantiate or destroy an object become held at the respective language operator, after checking whether or not *r* is paused. We call this set of held threads r_{ht}.

`getObjects` acquires a list of all existing objects that have been instantiated via *r*, giving the list r_{objs}. Because *r* is currently paused, it is assured that r_{objs} contains *all* objects whose implementations are (and ever will be) sourced from the component to which *r* is currently wired.

`rewire` changes the current wiring of *r* to point to the equivalent provided interface of *sinkCom*.

`resume` removes the paused status from *r* and allows the set of threads in r_{ht} to resume execution, thereby enabling any held object instantiation or destruction operations to proceed. After this point, any instantiation operators will resolve against the component to which *r* is now wired, rather than to its previous wiring.

Our adaptation protocol uses the above four operations on lines 4–7. The result is that the wiring graph at the component level has been adapted, illustrated in the left half of Fig. 4. The protocol is then left with a set of objects r_{objs} whose implementations belong to the previous wiring of *r*. To complete the adaptation, each such object must have its implementation updated to be from the current wiring of *r*. This procedure is performed in the loop from lines 8–16 and is illustrated on the right of Fig. 4.

`pauseObject` first checks if the given object has been destroyed by this point (recall that object destruction was re-enabled by `resume`). If not, this individual object's destruction is prevented by setting a flag on the object such that a destruction operator will be held until the flag is unset; `pauseObject` then returns `true`. The remaining set of operations on lines 10–14 can then proceed in the knowledge that the object they operate on will not be destroyed in the meantime. A successful invocation of `pauseObject` also prevents any new function calls from being made on the given object, holding any such calls at their invocation operator in a set o_{ht}.

`adaptConstruct` dynamically creates a new object from the component to which r is now wired (by `rewire` as described above). This object, a, is specially created in such a way that it shares the transfer state fields (if any exist) of $r_{objs[i]}$, instead of having its own fresh copy.

`rewireObject` changes the internal link of the proxy object to which $r_{objs[i]}$ refers (recall that all references to objects actually refer to their proxy object), pointing that link at a instead of its previous location. As a return value, rewireObject gives a different proxy object b, the internal link of which refers to the object implementation to which the proxy object of $r_{objs[i]}$ used to refer.

`resumeObject` allows any function calls held in o_{ht} to proceed into the object (whose implementation object is now one sourced from *sinkCom*), also allowing all future calls to immediately proceed into the object.

`waitForObject` blocks until all function calls currently operating within the given object complete.

`adaptDestroy` destroys the given object in such a way that its transfer state fields are not also destroyed.

For adaptations of stateless objects, used in cases when the corresponding interface has no state transfer fields, our adaptation protocol is arranged as in Alg. 1. For stateful objects, the `waitForObject` operator instead appears just before line 10 and is given $r_{objs[i]}$ as its parameter. This ensures that the previous implementation has finished any potential modifications to an object's transfer fields before any logic in the new implementation occurs.

2.1.3 Structuring for discoverable code

An emergent software system must be able to autonomously discover usable components for different parts of itself. We wanted to support this without any extra wiring specifications or manifest files, as are common in many component models [19, 16]. In doing so we avoid developers having to do any work beyond writing component functionality. Instead, we chose to make discoverable code an inherent feature of our platform.

Our solution here is simply to define a fixed structure for projects. The root folder of a project therefore contains a 'resources' directory tree containing the source code of all interface types, and a symmetrical directory tree containing all components that implement those interfaces. For a component that declares a required interface of type `io.File`, we then know to look in the directory '`io`' for all potential implementation components of this interface.

2.1.4 Interface to higher system layers

To higher layers of RE^X, Dana provides the following API: **load** and **unload** a component into or out of memory; **get** the set of interfaces (provided and required) of a component; **connect** a component's required interface to another component's provided interface (for initial system assembly); and **adapt** a component's required interface to connect to an equivalent provided interface on a different component (via the above adaptation protocol).

2.2 Perception, Assembly and Learning

Whereas Dana provides the fundamental mechanisms to build systems, our perception, assembly and learning framework (PAL) abstracts over entire systems for online learning. Specifically, PAL **assembles** sets of discovered components into working systems; **perceives** the health of those systems and the conditions of their deployment environment; and **learns** correlations between a system's health, its current environment conditions, and its current assembly. Each of these elements operates at runtime, while the target emergent software system is executing. Unlike existing work, we use no models or architectural representations of software [12, 23], instead enabling systems to emerge autonomously as a continuous process.

2.2.1 Assembly

The assembly module of our framework is responsible for discovering the possible units of logic (i.e., components) that can form a given system; assembling a particular configuration of those components to create a working system; and re-assembling the running system to a different configuration using our adaptation protocol.

The assembly module starts with a 'main component' of a target software system (such as that shown in Fig. 2). The required interfaces of this component are read, and all available components that offer compatible provided interfaces are then discovered. This is done using Dana's inherent structuring for discoverable code: the package path of each required interface is converted to a local directory path, which is scanned for any components that provide this interface type. For each component found, the required interfaces of *those* components are read, and further components are discovered with corresponding provided interfaces. This procedure continues recursively until a full set of possible system compositions is discovered, and can be re-run periodically to detect new components.

We expect there to be multiple different implementations of each provided interface because there are typically several ways to solve a given problem, such as the use of different memory cache replacement algorithms or different search algorithms. Each such variant offers equally valid functionality relative to a provided interface, but implementation differences imply that their respective performance characteristics will differ according to different input ranges or deployment environment characteristics that are encountered by the system.

When the discovery procedure is complete, the assembly module provides a list of strings, each of which is a full description of one configuration of components. The assembly module can then be instructed to assemble the target system into one such configuration. If the system is not yet assembled, this means simply loading each component into memory and connecting the appropriate required and provided interfaces together, then calling the main method of the main component to start the system.

If the target system is already assembled in a particular configuration, a command to re-assemble it into a different one uses our adaptation protocol to seamlessly shift to the alternative. In detail, starting from the main component of the target system, the assembly module walks through the inter-component wiring graph to discover the difference points between the current configuration and the new one. For each such difference, the corresponding alternative component is loaded into memory, along with all components (recursively) that it requires, and the adaptation protocol is used to adapt to that component. The old component (and components that it required and that are not in use by other parts of the system) is then removed.

2.2.2 Perception

The ability to assemble and re-assemble a software system into different configurations of components must be guided in some way. Key to this is an ability to perceive the way the software 'feels' at a given point in time, and the way the software's deployment environment 'looks' at correlated points in time. These streams of perception can then be mapped to the software's current assembly (i.e., the way it is behaving) to understand how different behaviors make the software feel in different environments.

This is achieved using our perception module. To enable perception throughout a system we use a `Recorder` interface. Any component can declare a required interface of this type and then use it to report one (or both) of two kinds of data as it sees fit: *events* and *metrics*.

Events represent the way individual software components are perceiving the outside world – their inputs or deployment environment conditions. Events have a standard structure, with a name, descriptor, and value. When an event is reported to a `Recorder`, a timestamp is added.

Metrics represent the way individual software components are perceiving themselves – how they 'feel'. Metrics again have a standard structure, including a name, a value, and a boolean flag indicating whether a high or low value of this metric is desirable. As with events, a timestamp is added to a metric when reported to a `Recorder`.

When a new configuration of the software system is selected via the assembly module, the perception module uses Dana's `getInterfaces` API to check the components of that configuration for any with a `Recorder` required interface. For all such components, their attached `Recorder` implementation component is periodically polled to collect any recently reported events or metrics, noting the component from which they originated.

2.2.3 Learning

Finally, our learning module is tasked with understanding the data from the perception module, and exploring different assembly configurations of the target system to understand how different behavior sets cause the software to react to different external stimuli. We describe the full details of our learning approach in the next section.

2.2.4 Interface to higher system layers

The perception and assembly modules provide the following API to the learning module: **setMain()**, selecting a 'main' component of a program to assemble; **getConfigs()**, returning a list of strings describing every possible configuration of components; **setConfig()**, taking a configuration string to assemble/re-assemble the system to; and **getPerception()**, returning all events and metrics that have been collected since this function was last called.

2.3 Linear bandits for rapid emergence

In this section we describe our approach to efficiently learning the correlations between perception of internal state and external environment, and the currently selected behavior of a system. We first define this problem more precisely with a case study of an emergent web server, and then we describe our learning approach in detail.

2.3.1 Problem definition

For our evaluation in this paper we use a web server as an example emergent software system. A partial structure of this is shown in Fig. 5, illustrating the set of possible configurations that each represent a valid system. For simplicity here we only show components that have variations – in reality, the set of components used to form this system is much larger, at over 30 components (of which only 15 are shown). The components not shown here include those for file system and socket operations, string handling utilities, abstract data type implementations, etc.

From this set of components, there are 42 possible assemblies in total, each of which results in a functional web server system but with differing behaviors. As examples, some such assemblies use a memory cache (of which there are several variants) while others use a compression algorithm; and some use a thread-per-client approach to concurrency while others use a thread-pool approach.

We must then establish which of these 42 options best suits the current external stimuli to which the software is being subjected. These external stimuli may also change, invalidating what has been learned to date and requiring further search iterations. An exhaustive search approach is clearly undesirable, causing the system to spend too long in sub-optimal configurations; we therefore need a way to balance exploration of untested parts of the search space with exploitation of solutions known to be good.

The components of our web server generate two kinds of perception data to inform this. *RequestHandler* implementations report a metric of their average response time to client requests, providing an internal perception of self. Implementations of the *HTTPHandler* interface report events of the resources being requested and their size. This represents the system's perception of its deployment environment. For each set of client request patterns that are input to the system, there then exists one composition of components (behavior) that optimizes the reward value

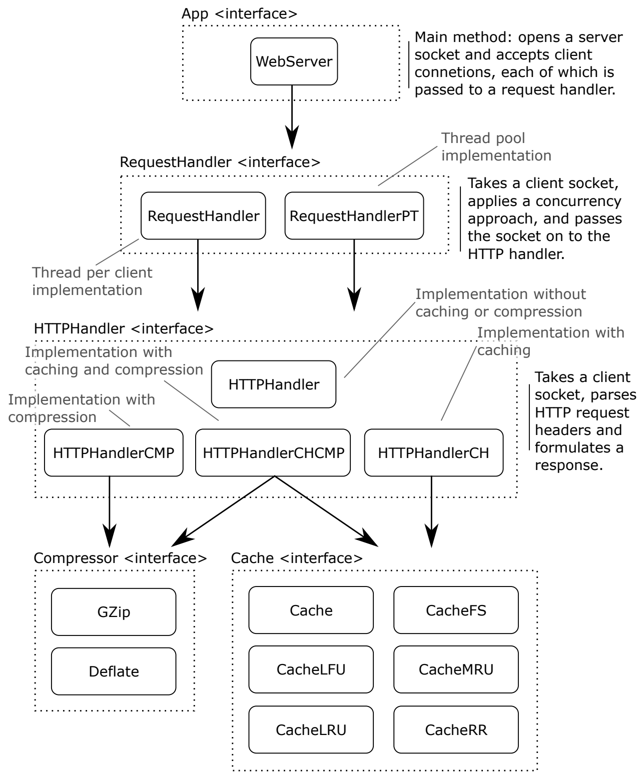

App <interface>
WebServer
Main method: opens a server socket and accepts client connetions, each of which is passed to a request handler.
RequestHandler <interface>
Thread pool implementation
RequestHandler
RequestHandlerPT
Takes a client socket, applies a concurrency approach, and passes the socket on to the HTTP handler.
Thread per client implementation
Implementation without caching or compression
HTTPHandler <interface>
Implementation with caching
Implementation with caching and compression
HTTPHandler
Takes a client socket, parses HTTP request headers and formulates a response.
Implementation with compression
HTTPHandlerCMP
HTTPHandlerCHCMP
HTTPHandlerCH
Compressor <interface>
Cache <interface>
GZip
Deflate
Cache
CacheFS
CacheLFU
CacheMRU
CacheLRU
CacheRR

Figure 5 – The set of components from which our web server can emerge. Boxes with dotted lines are interfaces, and those with solid lines are components implementing an interface. Arrows show required interfaces of particular components. The general purpose of each interface's implementations is noted by the interface, and a description of how the available implementation variations of that interface work is also indicated.

(i.e., minimizes response time) for a given environment. In future we expect multiple reward values (from different components) and dimensions of external perception (for example including resource levels of the host machine), but for now the above values are sufficient to explore the concept of runtime emergent software systems.

Our solution to this configuration search and learning problem is based on the statistical learning approach proposed by Scott [27]. In the remainder of this section we present the details of how we apply this approach.

We first cast our fundamental problem as a 'multi-armed bandit', for which the learning approach is intended. We then discuss the concept of Thompson sampling and how we use it to simultaneously update performance estimates of *multiple* configurations after experimenting with just one of them, and how we use Bayesian regression to derive beliefs about individual component performance within a configuration. Besides adapting the approach for our particular problem, we make two changes: (i) we use Bayesian linear regression instead of probit regression, enabling us to handle continuously distributed results; and (ii) we add a simple classifier system to provide memory of environment changes over time.

2.3.2 The Multi-armed Bandit Formulation

Online learning must balance the exploration of under-tested configurations with exploiting configurations already known to perform well [27]. The canonical form of this is the multi-armed bandit problem, devised for clinical trials [30, 7], and recently a dominant paradigm for optimization on the web [27, 9]. A multi-armed bandit has a set of available actions called arms. Each time an arm is chosen, a random reward is received, which depends (only) on the selected arm. The objective is to maximize the total reward obtained. While short-term reward is maximized by playing arms currently believed to have high reward, long-term benefit is maximized by exploring to ensure that we don't fail to find the best arm.

In the case of our emergent software, each possible configuration is considered an arm, and the reward given by playing an arm (i.e., selecting a particular configuration of the web server) is defined by our metrics (i.e., the average response time of this configuration to client requests).

One general method for tackling the multi-armed bandit problem is Thompson sampling. Theoretical performance guarantees exist for Thompson sampling in general settings [22, 5, 25], and the technique has been empirically shown to perform extremely well [9, 27]. The key feature of Thompson sampling is that each arm is played with the probability it is the best arm given the information to date. This requires the use of Bayesian inference to produce 'posterior distributions' that code our beliefs about unknown quantities of interest, in this case the expected values of the arms (see Sec. 2.3.3). With this inference, it has been shown that Thompson sampling can be efficiently implemented by drawing a single random sample from the posterior distribution of all unknown quantities, then selecting the arm which performs best conditional on this sampled value being the truth [22].

For example, suppose our unknown quantities are the expected value of each arm, and beliefs about these quantities are encoded as (posterior) probability distributions with densities given by bell curves. The center of the bell curve is then the average reward seen on that arm to date, and the spread is our level of uncertainty (with high uncertainty a result of few observations). For an arm to be selected with Thompson sampling, the random sample from its bell curve must be higher than corresponding samples from all other arms; to have a non-negligible probability of being selected, the distribution must be capable of producing high samples. This is true if either the center point is high or if the spread is large, corresponding respectively to high average observed rewards or high uncertainty.

The effect is that the arms most likely to be played are those that experience suggests are likely to perform well, and those that may perform well but we have insufficient information about. Arms for which we have good information that they will perform badly are played with very low probability. As more information is gained, and beliefs concentrate on the truth, no arms will remain for which there is insufficient information. Thus, in the long term, optimal arms are played with very high probability.

2.3.3 Forming beliefs

Thompson sampling balances exploration and exploitation in the presence of a Bayesian estimate of arm values. In traditional bandit settings, the value of each arm is estimated independently. However, the combinatorial explosion in the total number of available configurations renders this approach undesirable since each arm will need to be experimented with multiple times. Our emergent software example described above results in 42 such arms; and for example introducing just one further caching variation (see Fig. 5) would increase this to 48. As the complexity of an emergent software system grows, it quickly becomes undesirable to consider all configurations and test each one at runtime. Furthermore, estimating the performance of each configuration independently ignores the fact that many configurations share components, and so are likely to have related performance characteristics.

We therefore follow and use a regression framework based on classical experimental design to *share* information across the different arms available to us [27]. The intuition behind this scheme is that the performance of any configuration using the `HTTPHandlerCH` component is in some way informative for any other configuration involving that component. This is formalized by modelling the expected reward for a given configuration as a function of the components deployed within that configuration. In detail, we code each interface as a factor variable, with number of levels equal to the number of available components for that interface. Standard *dummy coding* is used, so that each level of the factor is compared to a fixed baseline level. For our web server example the effect of this is to model the expected reward of a configuration as

$$\begin{aligned}\beta_0 + &\beta_1 \mathbb{I}_{\text{RequestPT}} \\ &+\beta_2 \mathbb{I}_{\text{HttpCMP}} + \beta_3 \mathbb{I}_{\text{HttpCH}} + \beta_4 \mathbb{I}_{\text{HttpCHCMP}} \\ &+\beta_5 \mathbb{I}_{\text{Deflate}} \\ &+\beta_6 \mathbb{I}_{\text{CacheFS}} + \beta_7 \mathbb{I}_{\text{CacheLFU}} + \beta_8 \mathbb{I}_{\text{CacheMRU}} \\ &\quad + \beta_9 \mathbb{I}_{\text{CacheLRU}} + \beta_{10} \mathbb{I}_{\text{CacheRR}}\end{aligned} \tag{1}$$

where β_i are unknown real numbers to be estimated, and indicator functions $\mathbb{I}_X$ take value 1 if component X is used in the configuration, and 0 otherwise. Note that coefficients for `RequestHandler`, `HttpHandler`, `GZip` and `Cache` are implicitly coded as baseline levels for the factors, so the above coefficients are interpreted as deviations from this baseline performance. In other words, if for example all of HttpCMP, HttpCH and HttpCHCMP are set to 0, this implies the default `HttpHandler` is in use and its reward is encoded in β_0. The model in Equation 1 can be automatically derived by our learning module, and has only 11 elements to estimate, instead of 42.

The standard linear regression model assumes observed rewards are equal to expected rewards (1) plus a 'noise' term for un-modelled variability. If we denote the vector of binary indicator variables for a given configuration as

$$\begin{aligned}x_{\text{conf}} = (1, &\mathbb{I}_{\text{RequestPT}}, \mathbb{I}_{\text{HttpCMP}}, \mathbb{I}_{\text{HttpCH}}, \mathbb{I}_{\text{HttpCHCMP}}, \\ &\mathbb{I}_{\text{Deflate}}, \mathbb{I}_{\text{CacheFS}}, \mathbb{I}_{\text{CacheLFU}}, \mathbb{I}_{\text{CacheMRU}}, \\ &\mathbb{I}_{\text{CacheLRU}}, \mathbb{I}_{\text{CacheRR}}),\end{aligned}$$ [3]

with the vector of unknown coefficients denoted as $\beta = (\beta_0, \beta_1, \ldots, \beta_{10})$, and observed reward as y, the assumed model of linear regression is then that

$$y = x_{\text{conf}}\beta + \varepsilon,$$

where ε is a zero-mean Gaussian random value independent of all other observed quantities, with unknown variance σ^2. After observing multiple configurations and their rewards, we have a list of (x_{conf}, y) pairs; regression then finds the single β value which makes all $x_{\text{conf}}\beta$ values as close as possible to their relative observed y values.

The Bayesian approach to regression is used so that we can support Thompson sampling for action selection; specifically the Bayesian approach produces a posterior probability distribution over β and σ^2 as its output from which to then sample [24]. We use the standard conjugate prior distribution, with σ^2 having an inverse-gamma prior with parameters a_0 and b_0, and β having a multivariate Gaussian prior conditional on σ^2 with parameters $\tilde{\beta}$ and $\sigma^2\Lambda_0^{-1}$. The parameters of the prior are specified in Sec. 2.3.4. The posterior distribution of σ^2 is again an inverse-gamma distribution with parameters updated by the data, and the posterior for β conditional on σ^2 is a multivariate Gaussian distribution dependent on the data.

2.3.4 Implementation

The above approach is implemented in our learning module. This maintains information about the history of selected configurations and the rewards obtained. It also stores an $m \times k$ 'action matrix', where m is the number of valid configurations (in our case 42), and k is the number of unknown regression coefficients β_i (in our case 11). Each row corresponds to a valid configuration, and consists of the vector x_{conf} of indicators for the configuration. Multiplying this action matrix by a vector of coefficients β returns a vector of $x_{\text{conf}}\beta$ values, and thus simultaneously evaluates Equation 1 for all valid configurations.

This action matrix is used when selecting which configuration to deploy. A single β and σ^2 are sampled from the posterior distribution resulting from linear regression and β is then multiplied by the action matrix to get Thompson-sampled values for each arm. The configuration corresponding to the row with the highest resulting value is then chosen and deployed. After a ten second observation window, the resulting reward y is observed, and the (x_{conf}, y) pair is stored. The posterior distribution is then

[3]The initial 1 is included as the intercept term which multiplies β_0 and is present for all configurations.

updated before repeating the process. Pseudocode for this is given in Alg. 2, in which the formulae for sampling from the posterior is given in lines 8–13.

When initializing the system, appropriate values must be chosen for the prior parameters so that the algorithm explores sufficiently without immediately dismissing configurations. We choose $a_0 = 1$ to give a weakly informative prior distribution. b_0 is then chosen so that the range of values supported by the inverse-gamma(a_0, b_0) distribution includes the reward variance in the data; we choose b_0 such that the *a priori* most likely standard deviation, $\sqrt{b_0/2}$, is approximately equal to the expected standard deviation of reward. For β, we use a prior mean $\tilde{\beta}$ with all values except the first equal to zero, as it is unknown how each component affects the performance of the web server. The value of β_0 encodes the base performance of the server; optimistic prior beliefs that β_0 is higher than rewards we actually observe encourages initial exploration as, before a lot of data has been observed, the belief will remain that unexplored configurations have higher rewards than those that have been observed. Thus we take $\tilde{\beta}_0$ (the first component of $\tilde{\beta}$) to be slightly higher than the reward level we actually expect from the system. For Λ_0, the inverse of the prior covariance, we take a default weakly informative prior and set Λ_0 to be the identity matrix multiplied by a small constant value, equal to 0.1 throughout this article. The particular values of $\tilde{\beta}_0$ and b_0 used for our experiments are reported in Sec. 3.

2.3.5 Handling deployment environment changes

In a traditional multi-armed bandit problem the reward distributions of each arm, while unknown to the player, do not change their distribution over time. Thus, if the optimal arm is found, playing that arm forever carries no disadvantage. In a software system, however, the rewards of the respective arms (i.e., configurations) may change over time as the deployment environment of the system changes. In our example, if the request pattern experienced by the web server changes, then the effectiveness of a given configuration may diverge from current estimates. Without accommodating for this, when the request pattern changes the system must take time to first 'unlearn' what it knows about the effectiveness of the available configurations and their constituent components, and then learn new estimates. If the request pattern then reverts back to its old form, the entire procedure must be repeated.

To optimize this, we augment our algorithm with the ability to categorize request pattern features, and to update its estimates accordingly. However, automatically deriving such categorizations in real-time is itself a challenging problem. For this paper we manually define two features, based on how we presume they will affect the web server.

The first feature is *entropy*, describing the number of different resources requested in a given time frame. High entropy indicates many different resources, while

Algorithm 2 Learning Algorithm

```
 1: //matrix of all available $x_{\text{conf}}$ vectors (configurations)
 2: $actionMatrix = assembly.getConfigs()$
 3: $X = new\ Matrix()$ //list of observed $x_{\text{conf}}$'s to date
 4: $y = new\ Vector()$ //list of rewards seen for each X
 5: $n = 0$
 6: while $running$ do
 7:     //do linear regression & sample from posterior
 8:     $\Lambda = X^T X + \Lambda_0$
 9:     $\beta = \Lambda^{-1}(\Lambda_0 \tilde{\beta} + X^T y)$
10:     $a = a_0 + (n/2)$
11:     $b = b_0 + (y^T y + \tilde{\beta}^T \Lambda_0 \tilde{\beta} - \beta^T \Lambda \beta) \times 0.5$
12:     $\sigma^2 = new\ InverseGamma(a, b).sample()$
13:     $sample = new\ Normal(\beta, \sigma^2 \Lambda^{-1}).sample()$
14:
15:     //select the new configuration to use
16:     $i = \arg\max(actionMatrix * sample)$
17:     $assembly.setConfig(i)$
18:
19:     //wait for 10 seconds, then record observations
20:     $result = 1/perception.getAverageMetric()$
21:     add row $i$ of $actionMatrix$ as new row of $X$
22:     add $result$ as new element of $y$
23:     $n$++
24: end while
```

zero entropy indicates a single resource requested repeatedly. A pattern with low entropy, where many requests are the same, may benefit from configurations using a caching component, while for high entropy patterns caching would not help, and may even be detrimental.

The second feature is *text volume*, describing how much of the content requested in a given time frame was textual (i.e., HTML, CSS or other text-based content). A request pattern with high text content will likely be served better by a configuration that makes use of a compression component, as text is highly compressible, whereas a request pattern with high image or video content would waste resources by using compression and achieve little as a result.

We have implemented a simple pattern-matching module that observes the stream of events from our perception module and classifies them as follows: if one type of request (video, text, or image) makes up more than half of the requests in an observation window, it is assumed the request pattern has 'low' entropy, and otherwise 'high'. If more than half of the requests made in an observation window are for text items, it is assumed that the request pattern currently is 'high' text, otherwise 'low'.

To incorporate these environment features in our learning approach, we add terms to Equation 1 corresponding to these features, and also interaction terms between environmental indicators and components we believe to be relevant. In particular, we expect text volume to affect the benefit of compression, and entropy to affect the benefit of caching. Equation 1 is therefore modified to consist of the following indicators, each with a regression coefficient β_i:

$$(1, \mathbb{I}_{\text{RequestPT}}, \mathbb{I}_{\text{HiEnt}}, \mathbb{I}_{\text{HiTxt}}, \mathbb{I}_{\text{HttpCMP(LowTxt)}}, \mathbb{I}_{\text{HttpCMP(HiTxt)}}, \mathbb{I}_{\text{HttpCH(LowEnt)}}, \mathbb{I}_{\text{HttpCH(HiEnt)}}, \mathbb{I}_{\text{HttpCHCMP(LowTxt,LowEnt)}}, \mathbb{I}_{\text{HttpCHCMP(HiTxt,LowEnt)}}, \mathbb{I}_{\text{HttpCHCMP(LowTxt,HiEnt)}}, \mathbb{I}_{\text{HttpCHCMP(HiTxt,HiEnt)}}, \mathbb{I}_{\text{Deflate}}, \mathbb{I}_{\text{CacheFS}}, \mathbb{I}_{\text{CacheLFU}}, \mathbb{I}_{\text{CacheMRU}}, \mathbb{I}_{\text{CacheLRU}}, \mathbb{I}_{\text{CacheRR}}). \tag{2}$$

Adding the environment indicators, and splitting the indicators for different HTTP handlers by the environment, adds 7 extra regression coefficients. It also increases the number of possible 'configurations' to $42 \times 4 = 168$, as each configuration can now be observed in 4 environment states. After a configuration is deployed, the resulting vector x_{conf} of an observation window includes the environment indicators and interaction indicators (i.e., all the indicators in Equation 2). The linear regression proceeds as before, but with k increased so that we still have one regression coefficient per indicator (in this case $k = 18$).

When it is time to select an action, we sample a β value from our posterior distribution as before, and multiply the action matrix by β to give the predicted value. However, not all configurations are available to us, since some are determined by the environment. We make the simplifying assumption that the environmental context in the current time period will be the same as in the previous period, and restrict our configuration to those that correspond to that environment. It is plausible that a model of the evolution of the environment could be built and used to improve the prediction of values, but is beyond the scope of this paper.

The effect of this enhancement is that components' performance levels in different environments may be updated without having to forget information when the environment changes. We see the benefit of this in Sec. 3.3.

3 Experimental Evaluation

The goal of our evaluation is to investigate whether optimal designs of a software system emerge rapidly using real-time learning. Specifically, we evaluate our approach in three key ways. We first examine the speed with which runtime adaptation occurs. This helps to show the viability of emergent software at runtime, which may frequently adapt in exploration periods. Second, we manually analyze the different possible compositions of our web server as a baseline, demonstrating that different optimals exist in different operating environments as a result of micro-variation. This validates our emergent software approach. Third, we examine RE^{X} in operation, particularly the effectiveness of our online learning approach to discover optimal compositions of behavior in real time.

Our evaluation is conducted using a real, live implementation of the emergent web server described in Sec. 2.3.1, orchestrated by RE^{X}. We run our system on commodity rackmount servers, hosted in a production datacenter, of a similar design to many datacenters around the world. In particular we used servers with Intel Xeon Quad Core 3.60 GHz CPUs and 16 GB of RAM, running Ubuntu Server 14.04. Similar machines were used as clients when generating workloads for our system, where client machines were situated on a different subnet (in a different physical building) to the server machines.

	Average	Maximum	Minimum
setConfig (idle)	509.60 ms	615.00 ms	397.00 ms
setConfig (busy)	1350.32 ms	5811.00 ms	510.00 ms
pause/resume (idle)	8.50 μs	9.94 μs	7.81 μs
pause/resume (busy)	13.22 μs	31.21 μs	8.51 μs
pauseObject/resumeObject (idle)	4.51 μs	5.34 μs	3.84 μs
pauseObject/resumeObject (busy)	28.54 μs	387.17 μs	4.35 μs
components adapted in setConfig()	1.22	3.00	1.00

Table 1 – Adaptation speed measured in different ways, from full configuration changes to individual component adaptations.

All of our source code, with instructions on how to reproduce all results reported here, is available online at [4].

3.1 Adaptation characteristics

We use our highly adaptive Dana programming language (see Sec. 2.1) to support low-cost adaptation. This is a key enabler of emergent software systems, which must be able to experiment with various configurations and adapt to those configurations when appropriate during program execution. In this section we evaluate the time taken to perform runtime adaptation in detail.

We consider two factors in performing runtime adaptation: the overall time taken by RE^{X} to move from one complete configuration to another; and the time taken to perform a single adaptation between two components. For each test we perform 100 configuration changes (moving to each of our 42 configurations at least twice) with a 5 second gap between each configuration change. Across all tests we assume that any components needed are already loaded into memory and ready for use.

The first of the factors we consider, moving from one complete configuration to another, involves parsing a configuration string passed to `setConfig()`, verifying the validity of a configuration, and performing the staged adaptation procedure for each point at which the new configuration differs from the current one. The first two rows in Table 1 show the average, maximum and minimum time taken to do this across 100 tests. The first row, marked 'idle', shows results when the web server is given no workload, while the second row marked 'busy' shows results when the web server is given a workload that causes it to use 100% CPU capacity. This indicates that the use of `setConfig()` is generally slower when the web server is busy. There are two reasons for this: first, `setConfig()` is processed on the same physical machine as the system under its control and so is given less CPU time when that system is busy; and second, when more requests are in progress at the web server, the adaptation protocol must wait longer for in-progress cross-component calls to finish (i.e., at `waitForObject`).

We now examine the time taken to perform a single adaptation between two components, using the adaptation protocol described in Sec. 2.1.2. This reveals the time that a part of the web server is actually paused and will therefore delay performance of one or more of its tasks. This can happen for two reasons: the `pause` operation on a required interface, which temporarily prevents new objects from being instantiated until `resume` is called, or the `pauseObject` operation on a particular object, which temporarily prevents any new function calls being made into that object until `resumeObject` is called. For these results we first note the difference in time unit compared to the above: pause durations are on the order of *microseconds*. With an idle web server, pause durations are higher across `pause`/`resume` as the base complexity of these instructions is higher (in particular building the object list r_{objs}). Under load, however, pause durations are dominated by `pauseObject`/`resumeObject`, as the list of held inter-object threads o_{ht} grows quickly and must then be iterated over to release each one (see Sec. 2.1.2).

The final row in Table 1 shows the average, maximum and minimum number of adaptations made during one `setConfig()` operation, indicating how many of the above microsecond pauses will occur for our web server during a configuration change. This is very low in our system, with a maximum of just three adaptations. This indicates that system configuration changes during emergent software exploration – which occur at 10 second intervals under our learning algorithm – will be of low impact.

3.2 Manual analysis of divergent optimality

Our approach to emergent software uses small software components, with differing implementations of the same features such as varied cache replacement algorithms, to enable optimal software to emerge by trying differing combinations of these components at runtime.

In this section we validate this approach, in particular showing that there are different configurations of our web server with different performance profiles under different operating environment ranges – necessitating the need to switch between them in order to maintain optimality over time. We refer to this property as *divergent optimality*. To understand whether or not divergent optimal configurations exist for our web server, we run every possible configuration against various client workload patterns. We then examine the resulting performance of each configuration, measured at the server as the time between receiving a request and sending the last byte of the response.

Request pattern	**File size (b) [GZ]**	**Default**	**Caching**	**Caching & compression**
Text *low entropy*	156,983 [12,757]	11.94 ms	9.56 ms	**0.70 ms**
Text *low entropy*	82,628 [11,949]	4.05 ms	**0.60 ms**	0.66 ms
Text *low entropy*	3,869 [1,930]	1.18 ms	**0.59 ms**	0.63 ms
Image *low entropy*	1,671,167 [1,667,464]	160.81 ms	**150.72 ms**	154.42 ms
Image *low entropy*	84,760 [66,914]	4.02 ms	**0.66 ms**	0.74 ms
Image *low entropy*	4,001 [3,895]	1.22 ms	**0.55 ms**	0.62 ms
Text *high entropy*	156,983 [12,757]	19.27 ms	19.66 ms	**3.04 ms**
Text *high entropy*	82,628 [11,949]	4.61 ms	3.27 ms	**3.07 ms**
Text *high entropy*	3,869 [1,930]	**1.25 ms**	2.93 ms	2.52 ms
Image *high entropy*	1,671,167 [1,667,464]	**156.50 ms**	156.64 ms	157.66 ms
Image *high entropy*	84,760 [66,914]	4.48 ms	3.19 ms	**2.94 ms**
Image *high entropy*	4,001 [3,895]	**1.30 ms**	2.90 ms	2.67 ms

Table 2 – Results of different configurations under different request patterns, showing average response times. The standard deviation throughout these results is low, at around 0.2.

Our results are shown in Table 2, which lists the fastest configuration from each group of configurations (i.e., the fastest configuration that uses neither caching nor compression, the fastest that uses caching, and the fastest that uses both caching and compression). We do not show results for configurations that only use compression, as they reliably perform worst across all of our experiments.

First we subject all configurations to client request patterns with low entropy. We divide this into two subcategories: text-dominated and image-dominated. The results are shown in the top half of Table 2, which also shows the general size of the files being requested along with the compressed size of these files using the GZip algorithm. For almost all of these low entropy request patterns, configurations with caching perform best – marginally better than those with both caching and compression. On investigation, this is because our configurations that use both compression and caching actually use slightly more instructions to check if a compressed version of a file is in the cache (i.e., they append '.gz' to the resource name before checking if that resource is in the cache). In most cases this slight delay is larger than the added network delay of sending the uncompressed version of the file. When the compression win is big enough, however, as in the first row of Table 2, the reverse is true and the caching plus compression solution is faster.

Next we subject all configurations to client request patterns with high entropy; specifically patterns that cycle through a set of 20 popular files. The corresponding results are shown in the lower half of Table 2, again divided into text-dominated and image-dominated requests. Here we see a different picture: in half of the tests, configurations without caching or compression are fastest. This is because, in these cases, reading from the local disk is very slightly faster than searching the cache. On investigation, in cache implementations that use a hash table this is caused by collisions in the hash function which then result in a linear search of a hash table bucket, adding latency. However, this result is reversed for the other half of these results, when the average compression ratio of the files being requested is sufficiently high that the network bandwidth saving overtakes the cache search latency.

These results confirm there are different optimal configurations of components that can form our target system in different environments. Low entropy and high text conditions favor configurations with caching and compression; low entropy and low text conditions favor configurations with caching only; and high entropy conditions favor a mixture of configurations. The subtleties within these results, and the fact that issues such as disk/memory latency will vary across machines, further motivate a real-time, machine-learning-based solution to building software.

3.3 Learning evaluation

We now examine the efficacy of RE^X at discovering the above results, on a live system, starting from no information. While some of these results may be obvious to a human observer, we provide the first example of an autonomous system able to rapidly assemble a corresponding solution at runtime. Our learning system uses only events and metrics reported by components of the live web server (Sec. 2.3.1), alongside the ability to dynamically assemble different configurations of discovered components, to find the best course of action over time.

By default our learning approach tries to maximize $1/responseTime$. Accordingly, we configure our learning algorithm with $\tilde{\beta}_0 = 1$, which is larger than the reciprocal of the response times in Table 2. The standard deviations in this data are on the order of 0.2, and we thus set $b_0 = 0.1$ so these are on a similar scale to $\sqrt{b_0/2}$.

As a theoretical baseline learning comparison, consider an approach that tests each configuration once before selecting the one that performed best. This takes 42 testing iterations before there is a chance of reaching optimality (as there are 42 available configurations), even without any noise in the observations whereby a configuration may need to be tested multiple times. Assuming each such test takes 10 seconds to get an average response time, this means a total of 420 seconds (7 minutes) to reach optimality. If successful, our approach should perform significantly better than this baseline in most cases.

We use a range of request patterns to evaluate our emergent software system, starting with simpler patterns. Each experiment is repeated 1,000 times and the interquartile ranges plotted. Each graph is plotted as *regret*, which is calculated as $(1/responseTime)_{chosenAction} - (1/responseTime)_{optimalAction}$ for each point in time

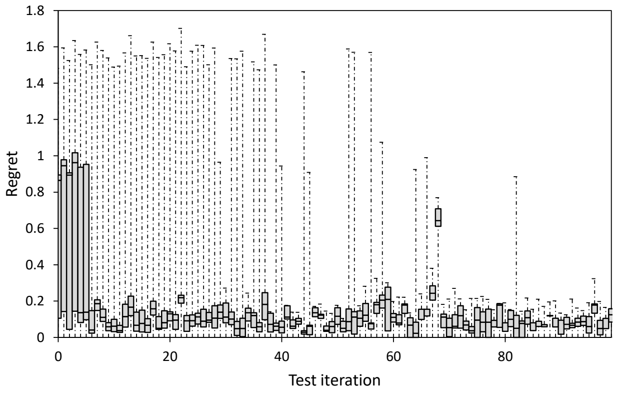


Figure 6 – Learning using response times to small text files.

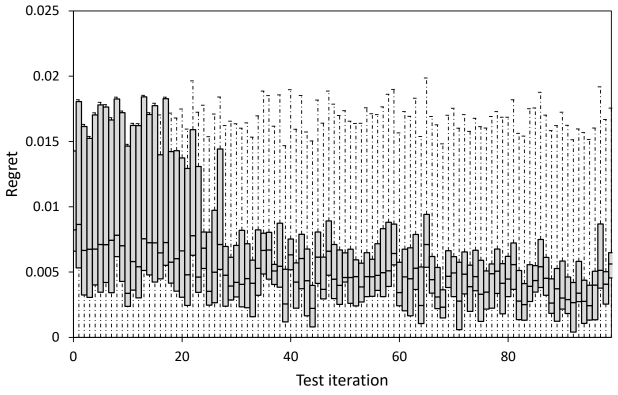
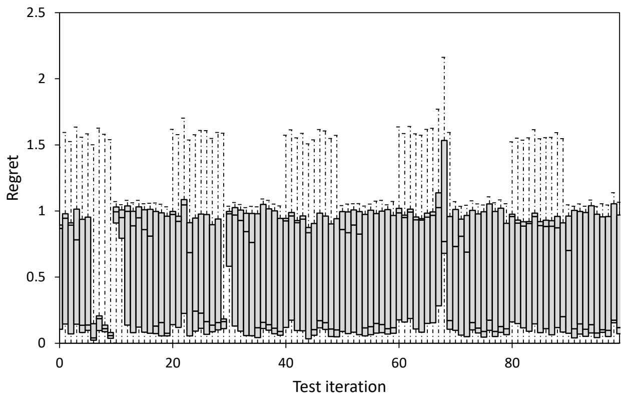


Figure 7 – Learning using response times to large text files, with adjusted prior values for $\tilde{\beta}_0$ and b_0.

(where knowledge of the optimal actions over time is based on our manual analysis of these request patterns). Specifically, the shaded boxes on these graphs show the size of the interquartile range from the distribution of regret results at each time step across all 1,000 experiments. The horizontal line dividing each shaded box is the median value. The whiskers above / below the shaded boxes show the highest / lowest results across all experiments.

Fig. 6 shows results for request patterns of small HTML files with low entropy. Here we see a dramatic reduction in regret after only a few iterations (where one 'iteration' represents a 10-second observation window). Although high regret is occasionally seen after this point, this is an inevitable artefact of continual exploration. Very good response times are learned here after just 50 seconds, which is significantly faster than the baseline described above. This demonstrates that our learning approach, based on estimating individual component contribution and then sharing information across all potential configurations, is very effective at avoiding exhaustive experimentation.

In Fig. 7 we show results for request patterns of large HTML files with low entropy. Here we see that the scale of our rewards has changed – i.e., the average response time for larger files is higher (almost 10 times) and as such our prior parameters were observed not to match



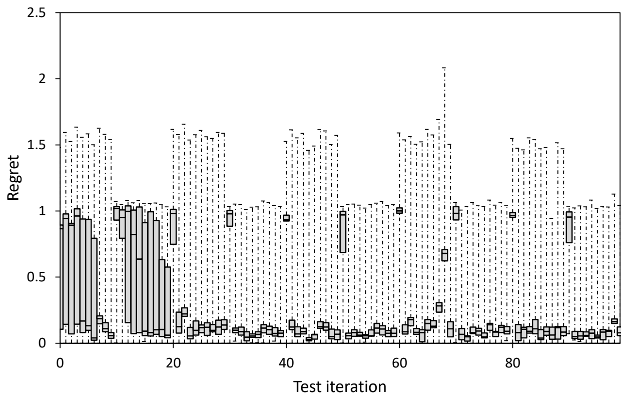


Figure 8 – Learning without (left) and with (right) categorization on a request pattern that changes every ten iterations.

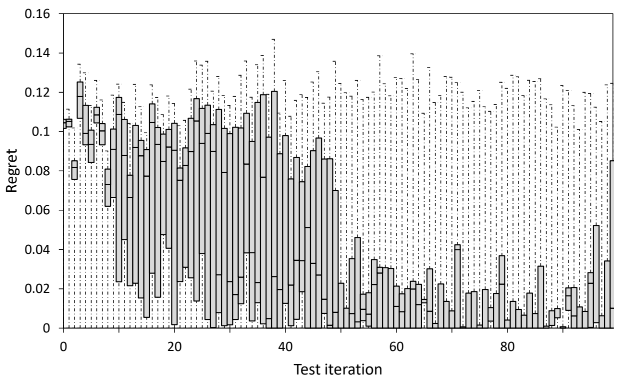


Figure 9 – Learning using response times to a realistic (and highly varying) request pattern, using the NASA server trace [2].

the data. For this experiment we therefore used adjusted prior parameters $\tilde{\beta}_0$ and b_0 that were each divided by 10 compared to the previous test. Again we see rapid convergence on an optimal software assembly, this time at around 20 iterations of the learning algorithm (roughly 200 seconds). The longer convergence time here is due to there being fewer samples from which to draw information (i.e., serving each request takes longer, providing less data per observation time window). Note that good prior values can easily be chosen automatically by sampling response times and calculating their mean and variance.

In Fig. 9 we show results from a real-world web server workload of highly mixed resource requests, taken from the publicly available NASA server trace [2]. This is a challenging request pattern due to its high variance over time, in which different kinds of resource are requested in very different volumes. As a result our learning approach finds it more difficult to compare like-for-like results as different configurations are tested. Initially regret here is generally high, but decreases steadily up to the 40th iteration mark. Overall the system still shows increased performance at least as well as our baseline.

Finally, we examine situations in which request patterns change between different characteristics of entropy and text volume, showing the ability of our platform to adjust to new external stimuli and remember historical information. This is demonstrated in Fig. 8, showing the results of tests in which the request pattern is alternated every ten iterations. When the system operates without categorization, shown on the left of Fig. 8, there is no clear change in regret as it must constantly 'forget' and 're-learn' estimates due to the shifting performance of configurations that it observes. However, with categorization the system exhibits learning behavior for the first two changes in request pattern, and then consistently makes low-regret choices despite the alternation between patterns. There is a brief increase in regret each time the request pattern changes, caused by the learning algorithm needing one observation window in which to observe the changes. This demonstrates that our addition of a simple pattern-matching system achieves the desired effect.

Overall, our results show rapid convergence on optimal software which emerge from online experimentation with different available configurations, with very little information about the nature of the target software system and the deployment conditions that it may experience. RE^X can be deployed on any hardware configuration (which may change the effects seen in Sec. 3.2), and in any deployment environment conditions, and will continually find the most effective system design. More broadly, RE^X can also show the rationale behind its choices to human developers, potentially leading to new development directions.

4 Related Work

While autonomic, self-adaptive and self-organizing computing are now well established, there is relatively little work at the level of autonomous runtime software composition (compared to a much larger body of work on autonomous parametric tuning). The majority of this work is model-driven – relying on substantial human-specification or offline training cycles, or using simple online heuristic search algorithms over carefully specified models. We survey the most closely related work below.

Grace *et al.* propose the use of human-specified adaptation policies to select between different communication interfaces in a river-monitoring scenario [17]. While the use of such adaptation policies is viable in simpler architectures, this becomes infeasible in more complex configurations where the set of component interactions is much larger. We therefore use an online learning approach to effectively discover the adaptation policy at runtime.

Chen *et al.* propose a weighted decision graph of service levels to generate model transformations in an online shopping system [10]. Wang *et al.*, meanwhile, propose a framework that exploits variability of software configurations for self-repair, using a goal model based on formal requirements [33]. By contrast we use a model-free approach in which components report their own current status from which we then infer global properties.

Bencomo *et al.* propose dynamic decision networks (a form of state machine), alongside a models-at-runtime approach to software composition, to decide at runtime between different network topologies for a remote data mirroring system based on perceived resilience levels [6]. This requires pre-specification of the decision network to determine configuration selection, rather than the online learning approach we take for emergent software.

Kouchnarenko and Weber propose the use of temporally-dependent logic to control software configuration, with a domain-specific notation to model temporal dependencies between adaptation actions in a self-driving vehicle control system [20]. While such temporal models may be a useful addition to constrain adaptation, they are again specified by human developers at design time rather than learned at runtime.

In FUSION [12], a feature-model framework is presented that uses offline training combined with online tuning to activate and deactivate selected feature modules at runtime (such as security or logging). Dynamic Software Product Lines [18] generalize the feature model approach as part of the software development process, typically using a pre-specified set of rules to trigger feature activation / deactivation at runtime. Our approach does not use a feature model, instead emerging a working system from a pool of components using online learning.

In SASSY [23], a self-adaptive architecture framework for service-oriented software is presented, using a set of models to describe software architecture and its QoS traits. Further work by Ewing and Menascé [13] applies a set of runtime heuristic search algorithms to the configuration search problem, including hill climbing and genetic algorithms. Our work differs in two ways: first we use a model-free approach, in which system composition is autonomously driven from a 'main' component; and second we apply a statistical machine learning approach to configuration search, based on sharing inferred per-component performance data across configurations.

Finally, we note that Thompson sampling with regression was first proposed by Scott [27] to select likely high performing versions of websites (i.e., with high vs. low quality images), updating beliefs on similar versions without needing to try each individually. We have applied this concept to runtime emergent software, but using Bayesian linear regression (rather than probit regression) to handle continuously distributed results, and a simple pattern matching approach to account for distinct workload patterns that cause different optimal software configurations.

5 Conclusion

Current approaches to self-adaptive software architectures require significant expertise in building models, policies and processes to define how and when software should adapt to its environment. We have presented a novel approach to runtime emergent software which avoids all such expertise, using purely machine-driven decisions about the assembly and adaptation of software. The result is to almost entirely remove human involvement in how self-adaptive systems behave, making this machine-led; and to produce systems that are responsive to the actual conditions that they encounter at runtime, and the way they perceive their behavior in these conditions.

Our approach has three major contributions that form our RE^X platform: a programming language for highly-adaptive assemblies of behaviors; a perception, assembly and learning framework to discover, monitor and control available assemblies; and a learning approach based on linear bandits that solves the resulting search space explosion by sharing information across assemblies.

Our results show that our approach is highly effective at rapidly discovering optimal compositions of behavior in a web server example, balancing exploration with exploitation, and is also highly responsive to changes in the software's deployment environment conditions over time.

In our future work we will broaden our approach to other types of application, and will also explore the automated generation of component variants, and further automation in environment classification. In the longer term we will continue to work towards shifting the system design paradigm even further into software itself – making software a leading member of its own development team.

Acknowledgements

This work was partially supported by the EPSRC *Deep Online Cognition* project, grant number EP/M029603/1. Roberto Rodrigues Filho would like to thank his sponsor, CAPES Brazil, for scholarship grant BEX 13292/13-7. The feedback of Prof. Gordon Blair and Dr. Sarah Clinch was very helpful in preparing this paper for submission. We also thank all of our reviewers for their thoughtful comments which helped to further improve the published version, and particularly our shepherd Dr. Petros Maniatis for his detailed feedback across several final iterations.

References

[1] Dana language: http://www.projectdana.com/.

[2] NASA web server trace: http://ita.ee.lbl.gov/html/contrib/NASA-HTTP.html.

[3] OSGI alliance: https://www.osgi.org/.

[4] Source code and experiments from this paper: http://research.projectdana.com/osdi2016porter.

[5] S. Agrawal and N. Goyal. Thompson sampling for contextual bandits with linear payoffs. *arXiv preprint arXiv:1209.3352*, 2012.

[6] N. Bencomo, A. Belaggoun, and V. Issarny. Dynamic decision networks for decision-making in self-adaptive systems: A case study. In *Software Engineering for Adaptive and Self-Managing Systems (SEAMS), 2013 ICSE Workshop on*, pages 113–122, May 2013.

[7] D. A. Berry and B. Fristedt. *Bandit problems: sequential allocation of experiments (Monographs on statistics and applied probability)*. Springer, 1985.

[8] E. Bruneton, T. Coupaye, M. Leclercq, V. Quema, and J.-B. Stefani. An open component model and its support in java. In *Component-Based Software Engineering*, volume 3054 of *LNCS*, pages 7–22. Springer Berlin Heidelberg, 2004.

[9] O. Chapelle and L. Li. An empirical evaluation of thompson sampling. In *Advances in neural information processing systems*, pages 2249–2257, 2011.

[10] B. Chen, X. Peng, Y. Yu, B. Nuseibeh, and W. Zhao. Self-adaptation through incremental generative model transformations at runtime. In *Proceedings of the 36th International Conference on Software Engineering*, ICSE 2014, pages 676–687, New York, NY, USA, 2014. ACM.

[11] G. Coulson, G. Blair, P. Grace, F. Taiani, A. Joolia, K. Lee, J. Ueyama, and T. Sivaharan. A generic component model for building systems software. *ACM Trans. on Comp. Systems*, 26(1):1:1–1:42, Mar. 2008.

[12] A. Elkhodary, N. Esfahani, and S. Malek. Fusion: A framework for engineering self-tuning self-adaptive software systems. In *Proceedings of the Eighteenth ACM SIGSOFT International Symposium on Foundations of Software Engineering*, FSE '10, pages 7–16, New York, NY, USA, 2010. ACM.

[13] J. M. Ewing and D. A. Menascé. A meta-controller method for improving run-time self-architecting in SOA systems. In *Proceedings of the 5th ACM/SPEC International Conference on Performance Engineering*, ICPE '14, pages 173–184, New York, NY, USA, 2014. ACM.

[14] F. Faniyi, P. R. Lewis, R. Bahsoon, and X. Yao. Architecting self-aware software systems. In *Software Architecture (WICSA), 2014 IEEE/IFIP Conference on*, pages 91–94, April 2014.

[15] A. G. Ganek and T. A. Corbi. The dawning of the autonomic computing era. *IBM Syst. J.*, 42(1):5–18, Jan. 2003.

[16] D. Gay, P. Levis, R. von Behren, M. Welsh, E. Brewer, and D. Culler. The nesc language: A holistic approach to networked embedded systems. In *Proceedings of the ACM SIGPLAN 2003 Conference on Programming Language Design and Implementation*, PLDI '03, pages 1–11, New York, NY, USA, 2003. ACM.

[17] P. Grace, D. Hughes, B. Porter, G. Blair, G. Coulson, and F. Taiani. Experiences with open overlays: a middleware approach to network heterogeneity. In *Proceedings of the 3rd ACM SIGOPS/EuroSys European Conference on Computer Systems 2008*, pages 123–136, April 2008.

[18] M. Hinchey, S. Park, and K. Schmid. Building dynamic software product lines. *Computer*, 45(10):22–26, Oct 2012.

[19] E. Kohler, R. Morris, B. Chen, J. Jannotti, and M. F. Kaashoek. The click modular router. *ACM Trans. Comput. Syst.*, 18(3):263–297, Aug. 2000.

[20] O. Kouchnarenko and J.-F. Weber. Adapting component-based systems at runtime via policies with temporal patterns. In *Formal Aspects of Component Software*, pages 234–253. Springer, 2014.

[21] J. Kramer and J. Magee. The evolving philosophers problem: dynamic change management. *IEEE Transactions on Software Engineering*, 16(11):1293–1306, 1990.

[22] B. C. May, N. Korda, A. Lee, and D. S. Leslie. Optimistic bayesian sampling in contextual-bandit problems. *The Journal of Machine Learning Research*, 13(1):2069–2106, 2012.

[23] D. Menascé, H. Gomaa, S. Malek, and J. Sousa. SASSY: A Framework for Self-Architecting Service-Oriented Systems. *Software, IEEE*, 28(6):78–85, Nov 2011.

[24] A. O'Hagan. *Kendall's Advanced Theory of Statistics: Bayesian inference. vol. 2B*. Number v. 2, pt. 2 in Kendall's library of statistics. Edward Arnold, 1994.

[25] D. Russo and B. Van Roy. Learning to optimize via posterior sampling. *Mathematics of Operations Research*, 39(4):1221–1243, 2014.

[26] M. Salehie and L. Tahvildari. Self-adaptive software: Landscape and research challenges. *ACM Transactions on Autonomous and Adaptive Systems (TAAS)*, 4(2):14, 2009.

[27] S. L. Scott. A modern bayesian look at the multi-armed bandit. *Applied Stochastic Models in Business and Industry*, 26(6):639–658, 2010.

[28] C. Soules, J. Appavoo, K. Hui, R. Wisniewski, D. Da Silva, G. Ganger, O. Krieger, M. Stumm, M. Auslander, M. Ostrowski, B. Rosenburg, and J. Xenidis. System support for online reconfiguration. In *Proceedings of the USENIX Annual Technical Conference*, pages 141–154, June 2003.

[29] C. Szyperski, D. Gruntz, and S. Murer. *Component Software: Beyond Object-Oriented Programming*. Acm Press Series. ACM Press, 2002.

[30] W. R. Thompson. On the Likelihood that one unknown probability exceeds another in view of the evidence of two samples. *Biometrika*, 25(3-4):285–294, 1933.

[31] S. Tomforde, J. Hähner, and C. Müller-Schloer. Incremental design of organic computing systems - moving system design from design-time to runtime. In *Proceedings of the 10th International Conference on Informatics in Control, Automation and Robotics*, pages 185–192, 2013.

[32] Y. Vandewoude, P. Ebraert, Y. Berbers, and T. D'Hondt. Tranquility: A low disruptive alternative to quiescence for ensuring safe dynamic updates. *IEEE Transactions on Software Engineering*, 33(12):856–868, 2007.

[33] Y. Wang and J. Mylopoulos. Self-repair through reconfiguration: A requirements engineering approach. In *Proceedings of the 2009 IEEE/ACM International Conference on Automated Software Engineering*, pages 257–268. IEEE Computer Society, 2009.

Yak: A High-Performance Big-Data-Friendly Garbage Collector

Khanh Nguyen[†] Lu Fang[†] Guoqing Xu[†] Brian Demsky[†]
Shan Lu[‡] Sanazsadat Alamian[†] Onur Mutlu[§]
University of California, Irvine[†] *University of Chicago*[‡] *ETH Zürich*[§]

Abstract

Most "Big Data" systems are written in managed languages, such as Java, C#, or Scala. These systems suffer from severe memory problems due to the massive volume of objects created to process input data. Allocating and deallocating a sea of data objects puts a severe strain on existing garbage collectors (GC), leading to high memory management overheads and reduced performance.

This paper describes the design and implementation of Yak, a "Big Data" friendly garbage collector that provides high throughput and low latency for *all* JVM-based languages. Yak divides the managed heap into a control space (CS) and a data space (DS), based on the observation that a typical data-intensive system has a clear distinction between a control path and a data path. Objects created in the control path are allocated in the CS and subject to regular tracing GC. The lifetimes of objects in the data path often align with *epochs* creating them. They are thus allocated in the DS and subject to region-based memory management. Our evaluation with three large systems shows very positive results.

1 Introduction

It is clear that Big Data analytics has become a key component of modern computing. Popular data processing frameworks such as Hadoop [4], Spark [67], Naiad [48], or Hyracks [12] are all developed in managed languages, such as Java, C#, or Scala, primarily because these languages 1) enable fast development cycles and 2) provide abundant library suites and community support.

However, managed languages come at a cost [36, 37, 39, 47, 51, 59, 60, 61, 62, 63]: memory management in Big Data systems is often prohibitively expensive. For example, garbage collection (GC) can account for close to 50% of the execution time of these systems [15, 23, 49, 50], severely damaging system performance. The problem becomes increasingly painful in latency-sensitive distributed cloud applications where long GC pause times on one node can make many/all other nodes wait, potentially delaying the processing of user requests for an unacceptably long time [43, 44].

Multiple factors contribute to slow GC execution. An obvious one is the massive volume of objects created by Big Data systems at run time. Recent techniques propose to move a large portion of these objects outside the managed heap [28, 50]. Such techniques can significantly reduce GC overhead, but inevitably substantially increase the burden on developers by requiring them to manage the non-garbage-collected memory, which negates much of the benefit of using managed languages.

A critical reason for slow GC execution is that object characteristics in Big Data systems do *not* match the heuristics employed by state-of-the-art GC algorithms. This issue could potentially be alleviated if we can design a more suitable GC algorithm for Big Data systems. Intelligently adapting the heuristics of GC to object characteristics of Big Data systems can enable efficient handling of the large volume of objects in Big Data systems without relinquishing the benefits of managed languages. This is a promising yet challenging approach that has not been explored in the past, and we explore it in this work.

1.1 Challenges and Opportunities

Two Paths, Two Hypotheses The key characteristics of heap objects in Big Data systems can be summarized as *two paths, two hypotheses*.

Evidence [15, 28, 50] shows that a typical data processing framework often has a clear logical distinction between a *control path* and a *data path*. As exemplified by Figure 1, the control path performs cluster management and scheduling, establishes communication channels between nodes, and interacts with users to parse queries and return results. The data path primarily consists of data manipulation functions that can be connected to form a data processing pipeline. Examples include data partitioners, built-in operations such as Join or Aggregate, and user-defined data functions such as Map or Reduce.

These two paths follow different heap usage patterns. On the one hand, the behavior of the control path is similar to that of conventional programs: it has a complicated logic, but it does not create many objects. Those created objects usually follow the *generational hypothesis*: most recently allocated objects are also most likely to become unreachable quickly; most objects have short life spans.

On the other hand, the data path, while simple in code logic, is the main source of object creation. And, objects created by it do *not* follow the generational hypothesis. Previous work [15] reports that more than 95% of the objects in Giraph [3] are created in supersteps that represent graph data with `Edge` and `Vertex` objects. The

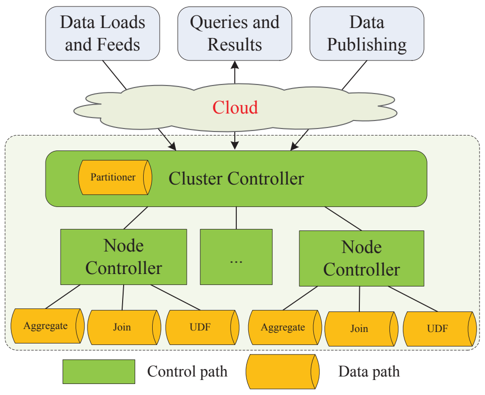


Figure 1: Graphical illustration of control and data paths.

execution of the data path often exhibits strong *epochal behavior* — each piece of data manipulation code is repeatedly executed. The execution of each epoch starts with allocating many objects to represent its input data and then manipulating them. These objects are often held in large arrays and stay alive throughout the epoch (*cf.* §3), which is often *not* a short period of time.

State-of-the-art GC State-of-the-art garbage collection algorithms, such as generational GC, collect the heap based on the *generational hypothesis*. The GC splits objects into a young and an old generation. Objects are initially allocated in the young generation. When a *nursery* GC runs, it identifies all young-generation objects that are reachable from the old generation, promotes them to the old generation, and then reclaims the entire young generation. Garbage collection for the old generation occurs infrequently. As long as the generational hypothesis holds, which is true for many large conventional applications that make heavy use of short-lived temporary data structures, generational GCs are efficient: a small number of objects *escape* to the old generation, and hence, most GC runs need to traverse only a small portion of the heap to identify and copy these escaping objects.

The Hypothesis Mismatch We find that, while the generational hypothesis holds for the control path of a data-intensive application, it does *not* match the epochal behavior of the data path, where *most* objects are created.

This mismatch leads to the fundamental challenge encountered by state-of-the-art GCs in data-intensive applications. Since newly-created objects often do *not* have short life spans, most GC runs spend significant time for identifying and moving young-generation objects into the old generation, while reclaiming little memory space. As an example, in GraphChi [41], a disk-based graph processing system, graph data in the *shard* defined by a vertex interval is first loaded into memory in each iteration, followed by the creation of many vertex objects to represent the data. These objects are *long-lived* and frequently visited to perform vertex updates. They cannot be reclaimed until the next vertex interval is processed. There can be dozens to hundreds of GC runs in each interval. Unfortunately, these runs end up moving *most* objects to the old generation and scanning almost the *entire* heap, while reclaiming *little* memory.

The epochal behavior of the data path also points to an opportunity not leveraged by existing GC algorithms – many data-path objects have the same life span and can be reclaimed together at the end of an epoch. We call this the *epochal hypothesis*. This hypothesis has been leveraged in *region-based memory management* [1, 8, 14, 25, 26, 28, 29, 30, 32, 40, 49, 50, 58], where objects created in an *epoch* are allocated in a memory region and efficiently deallocated as a whole when the epoch ends.

Unfortunately, existing region-based techniques need either sophisticated static analyses [1, 8, 14, 25, 26, 28, 29], which cannot scale to large systems, or heavy manual refactoring [28, 50], to guarantee that objects created in an epoch are indeed unreachable at the end of the epoch. Hence, such techniques have not been part of any garbage collector, to our knowledge.

1.2 Our Solution: The Yak GC

This paper presents Yak,[1] a high-throughput, low-latency GC tailored for managed Big Data systems. While GC has been extensively studied, existing research centers around the generational hypothesis, improving various aspects of the collection/application performance based on this hypothesis. Yak, in contrast, tailors the GC algorithm to the two very different types of object behavior (generational and epochal) observed in modern data-intensive workloads. Yak is the *first hybrid GC* that splits the heap into a control space (CS) and a data space (DS), which respectively employ generation-based and region-based algorithms to automatically manage memory.

Yak requires the developer to mark the beginning and end points of each epoch in the program. This is a simple task that even novices can do in minutes, and is already required by many Big Data infrastructures (*e.g.*, the `setup/cleanup` APIs in Hadoop [4]). Objects created inside each epoch are allocated in the DS, while those created outside are allocated in the CS. Since the number of objects to be traced in the CS is very small and only escaping objects in the DS need tracing, the memory management cost can be substantially reduced compared to a state-of-the-art generational GC.

While the idea appears simple, there are many challenges in developing a practical solution. First, we need to make the two styles of heap management for CS and DS smoothly co-exist inside one GC. For example, the generational collector that manages the CS in normal

[1] Yak is a wild ox that digests food with multiple stomachs.

ways should ignore some outgoing references to avoid getting in the way of DS management, and also keep track of incoming references to avoid deallocating CS objects referenced by DS objects (§5.4).

Second, we need to manage DS regions correctly. That is, we need to correctly handle the small number of objects that are allocated inside an epoch but escape to either other epochs or the control path. Naïvely deallocating the entire region for an epoch when the epoch ends can cause program failures. This is exactly the challenge encountered by past region-based memory management techniques.

Existing Big Data memory-management systems, such as Facade [50] and Broom [28], require developers to manually refactor both user and system programs to take control objects out of the data path, which, in turn, requires a deep understanding of the life spans of *all objects* created in the data path. This is a difficult task, which can take experienced developers weeks of effort or even longer. It essentially brings back the burden of manual memory management that managed languages freed developers from, imposing substantial practical limitations.

Yak offers an *automated and systematic solution*, requiring *zero* code refactoring. Yak allocates all objects created in an epoch in the DS, automatically tracks and identifies all escaping objects, and then uses a *promotion algorithm* to migrate escaping objects during region deallocation. This handling completely frees the developers from the stress of understanding object life spans, making Yak practical enough to be used in real settings (§5).

Third, we need to manage the DS region efficiently. This includes efficiently tracking escaping objects and migrating them. Naïvely monitoring every heap access to track escaping objects would lead to prohibitive overhead. Instead, we require light checking only before a heap write, but *not* on any heap read (§5.2). To guarantee memory correctness (*i.e.*, no live object deallocation), Yak also employs a lightweight "stop-the-world" treatment when a region is deallocated, without introducing significant stalls (§5.3).

1.3 Summary of Results

We implemented Yak inside Oracle's production JVM, OpenJDK 8. The JVM-based implementation enables Yak to work for all JVM-based languages, such as Java, Python, or Scala, while systems such as Facade [50] and Broom [28] work only for the specific languages they are designed for. We have evaluated Yak on three popular frameworks, *i.e.*, Hyracks [12], Hadoop [4], and GraphChi [41], with various types of applications and workloads. Our results show that Yak reduces GC latency by 1.4 – 44.3× and improves overall application performance by 12.5% – 7.2×, compared to the default Parallel Scavenge production GC in the JVM.

2 Related Work

Garbage Collection *Tracing garbage collectors* are the mainstream collectors in modern systems. A tracing GC performs allocation of new objects, identification of live objects, and reclamation of free memory. It traces live objects by following references, starting from a set of root objects that are directly reachable from live stack variables and global variables. It computes a *transitive closure* of live objects; objects that are unreachable during tracing are guaranteed to be dead and will be reclaimed.

There are four kinds of canonical tracing collectors: *mark-sweep*, *mark-region*, *semi-space*, and *mark-compact*. They all identify live objects the same way as discussed above. Their allocation and reclamation strategies differ significantly. Mark-sweep collectors allocate from a free list, mark live objects, and then put reclaimed memory back on the free list [24, 46]. Since a mark-sweep collector does not move live objects, it is time- and space-efficient, but it sacrifices locality for contemporaneously allocated objects. Mark-region collectors [7, 11, 13] reclaim contiguous free regions to provide contiguous allocation. Some mark-region collectors such as Immix [11] can also reduce fragmentation by mixing copying and marking. Semi-space [5, 6, 10, 17, 22, 34, 56] and mark-compact [19, 38, 55] collectors both move live objects. They put contemporaneously-allocated objects next to each other in a space, providing good locality.

These canonical algorithms serve as building blocks for more sophisticated algorithms such as the generational GC (*e.g.*, [56]), which divides the heap into a young and an old generation. Most GC runs are *nursery (minor) collections* that only scan references from the old to the young generation, move reachable objects into the old generation, and then free the entire young generation. When nursery GCs are not effective, a *full-heap (major) collection* scans both generations.

At first glance, Yak is similar to generational GC in that it promotes objects reachable after an epoch and then frees the entire epoch region. However, the regions in Yak have completely different and much richer semantics than the two generations in a generational GC. Consequently, Yak encounters completely different challenges and uses a design that is different from a generational GC. Specifically, in Yak, regions are thread-private; they reflect nested epochs; many regions could exist at any single moment. Therefore, to efficiently check which objects are escaping, we cannot rely on a traditional tracing algorithm; escaping objects may have multiple destination regions, instead of just the single old generation.

Connectivity-based garbage collection (CBGC) [33] is a family of algorithms that place objects into partitions by performing connectivity analyses on the object graph. A connectivity analysis can be based on types, allocations, or the partitioning introduced by Harris [31]. Garbage

First (G1) [22] is a generational algorithm that divides the heap into many small regions and gives higher collection priority to regions with more garbage. While CBGC, G1, and Yak each uses a notion of *region*, each has completely different semantics for the region and hence a different design. For example, objects inside a G1 region are *not* expected to have lifespans that are similar to each other.

Region-based Memory Management Region-based memory management was first used in the implementations of functional languages [1, 58] such as Standard ML [30], and then was extended to Prolog [45], C [25, 26, 29, 32], Java [18, 54], as well as real-time Java [8, 14, 40]. Existing region-based techniques rely heavily on *static analyses*. Unfortunately, these analyses either examine the whole program to identify region-allocable objects, which cannot scale to Big Data systems that all have large codebases, or require developers to use a brand new programming model, such as region types [8, 14]. In contrast, Yak is a pure dynamic technique that easily scales to large systems and requires only straightforward marking of epochs from users.

Big Data Memory Optimizations A variety of data computation models and processing systems have been developed in the past decade [4, 12, 16, 20, 21, 35, 52, 53, 57, 64, 65, 66, 67]. All of these frameworks were developed in managed languages and can benefit immediately from Yak, as demonstrated in our evaluation (*cf.* §6).

Bu et al. studied several data processing systems [15] and showed that a "bloat-free" design (*i.e.*, no objects allowed in data processing units), which is unfortunately impractical in modern Big Data systems, can make the system orders of magnitude more scalable.

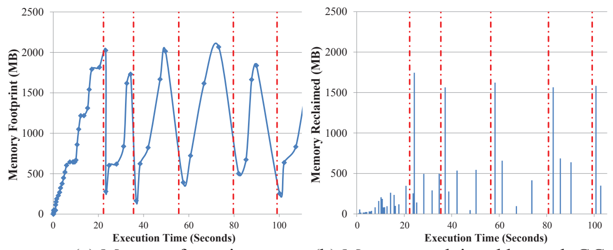
This insight has inspired recent work, like Facade [50], Broom [28], lifetime-based memory management [42], as well as Yak. Facade allocates data items into native memory pages that are deallocated in batch. Broom aims to replace the GC system by using regions with different scopes to manipulate objects with similar lifetimes. While promising, they both require extensive programmer intervention, as they move most objects out of the managed heap. For example, users must annotate the code and determine "data classes" and "boundary classes" to use Facade or explicitly use Broom APIs to allocate objects in regions. Yak is designed to free developers from the burden of understanding object lifetimes to use regions, making region-based memory management part of the managed runtime.

NumaGiC [27] is a new GC for "Big Data" on NUMA machines. It considers data location when performing (de-)allocation. However, as a generational GC, NumaGiC shares with modern GCs the problems discussed in §1.

Another orthogonal line of research on reducing GC pauses is building a holistic runtime for distributed Big Data systems [43, 44]. The runtime collectively manages the heap on different nodes, coordinating GC pauses to make them occur at times that are convenient for applications. Different from these techniques, Yak focuses on improving per-node memory management efficiency.

3 Motivation

We have conducted several experiments to validate our epochal hypothesis. Figure 2 depicts the memory footprint and its correlation with epochs when PageRank was executed on GraphChi to process a sample of the twitter-2010 graph (with 100M edges) on a server machine with 2 Intel(R) Xeon(R) CPU E5-2630 v2 processors running CentOS 6.6. We used the state-of-the-art Parallel Scavenge GC. In GraphChi, we defined an epoch as the processing of a sub-interval. While GraphChi uses multiple threads to perform vertex updates in each sub-interval, different sub-intervals are processed sequentially.

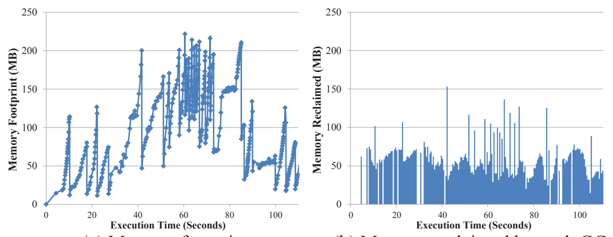


(a) Memory footprint (b) Memory reclaimed by each GC

Figure 2: Memory footprint for GraphChi [41] execution (GC consumes 73% of run time). Each dot in (a) represents the memory consumption measured right after a GC; each bar in (b) shows how much memory is reclaimed by a GC; dotted vertical lines show the epoch boundaries.

In the GraphChi experiment, GC takes 73% of run time. Each epoch lasts about 20 seconds, denoted by dotted lines in Figure 2. We can observe clear correlation between the end point of each epoch and each significant memory drop (Figure 2 (a)) as well as each large memory reclamation (Figure 2 (b)). During each epoch, many GC runs occur and each reclaims little memory (Figure 2 (b)).

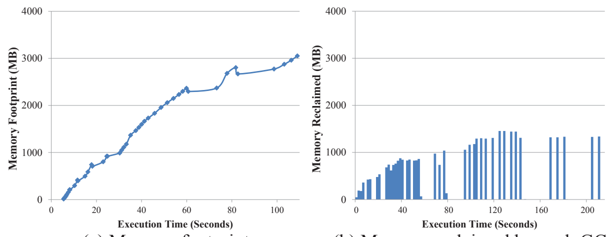
For comparison, we also measured the memory usage of programs in the DaCapo benchmark suite [9], widely-used for evaluating JVM techniques. Figure 3 shows the memory footprint of Eclipse under large workloads provided by DaCapo. Eclipse is a popular development IDE and compiler frontend. It is an example of applications that have complex logic but process small amounts of data. GC performs well for Eclipse, taking only 2.4% of total execution time and reclaiming significant memory in each GC run. We do not observe epochal patterns in Figure 3. While other DaCapo benchmarks may exhibit some epochal behavior, epochs in these programs are often not clearly defined and finding them is not easy for *application developers* who are not familiar with the *system codebase*.



(a) Memory footprint (b) Memory reclaimed by each GC

Figure 3: Eclipse execution (GC takes 2.4% of time).

Strawman Can we solve the problem by forcing GC runs to happen *only* at the end of epochs? This simple approach would not work due to the multi-threaded nature of real systems. In systems like GraphChi, each epoch spawns many threads that collectively consume a huge amount of memory. Waiting until the end of an epoch to conduct GC could easily cause out-of-memory crashes. In systems like Hyracks [12], a distributed dataflow engine, different threads have various processing speeds and reach epoch ends at different times. Invoking the GC when one thread finishes an epoch would still make the GC traverse many live objects created by other threads, leading to wasted effort. This problem is illustrated in Figure 4, which shows memory footprint of one slave node when Hyracks performs word counting over a 14GB text dataset on an 11-node cluster. Each node was configured to run multiple Map and Reduce workers and have a 12GB heap. There are no epochal patterns in the figure, exactly because many worker threads execute in parallel and reach the end of an epoch at different times.



(a) Memory footprint (b) Memory reclaimed by each GC

Figure 4: Hyracks WordCount (GC takes 33.6% of time).

4 Design Overview

The overall idea of Yak is to split the heap into a conventional CS and a region-based DS, and use different mechanisms to manage them.

When to Create & Deallocate DS Regions? A region is created (deallocated) in the DS whenever an epoch starts (ends). This region holds *all* objects created inside the epoch. An epoch is the execution of a block of data transformation code. Note that the notion of an epoch is well-defined in Big Data systems. For example, in Hyracks [12], the body of a dataflow operator is enclosed by calls to `open` and `close`. Similarly, a user-defined (Map/Reduce) task in Hadoop [4] is enclosed by calls to `setup` and `cleanup`.

To enable a unified treatment across different Big Data systems, Yak expects a pair of user annotations, *epoch_start* and *epoch_end*. These annotations are translated into two native function calls at run time to inform the JVM of the start/end of an epoch. Placing these annotations requires negligible manual effort. Even a novice, without much knowledge about the system, can easily find and annotate epochs in a few minutes. Yak guarantees execution correctness regardless of where epoch annotations are placed. Of course, the locations of epoch boundaries do affect performance: if objects in a designated epoch have very different life spans, many of them need to be copied when the epoch ends, creating overhead.

In practice, we need to consider a few more issues about the epoch concept. One is the *nested relationships* exhibited by epochs in real systems. A typical example is GraphChi [41], where a computational iteration naturally represents an epoch. Each iteration iteratively loads and processes all shards, and hence, the loading and processing of each memory shard (called *interval* in GraphChi) forms a *sub-epoch* inside the computational iteration. Since a shard is often too large to be loaded entirely into memory, GraphChi further breaks it into several *sub-intervals*, each of which forms a *sub-sub-epoch*.

Yak supports *nested regions* for performance benefits – unreachable objects inside an inner epoch can be reclaimed long before an outer epoch ends, preventing the memory footprint from aggressively growing. Specifically, if an *epoch_start* is encountered in the middle of an already-running epoch, a sub-epoch starts; subsequently a new region is created, and considered a child of the existing region. All subsequent object allocations take place in the child region until an *epoch_end* is seen. We do not place any restrictions on regions; objects in arbitrary regions are allowed to mutually reference one another.

The other issue is how to create regions when multiple threads execute the same piece of data-processing code concurrently. We could allow those threads to share one region. However, this would introduce complicated thread-synchronization problems; and might also delay memory recycling when multiple threads exit the epoch at different times, causing memory pressure. Yak creates one region for each dynamic instance of an epoch. When two threads execute the same piece of epoch code, they each get their own regions without having to worry about synchronization.

Overall, at any moment of execution, multiple epochs and hence regions could exist. They can be partially ordered based on their nesting relationships, forming a semilattice structure. As shown in Figure 5, each node on the semilattice is a region of form $\langle r_{ij}, t_k \rangle$, where r_{ij} denotes the j-th execution of epoch r_i and t_k denotes the thread executing the epoch. For example, region $\langle r_{21}, t_1 \rangle$ is a child of $\langle r_{11}, t_1 \rangle$, because epoch r_2 is nested in epoch r_1 in the program and they are executed by the same thread t_1. Two regions (*e.g.*, $\langle r_{11}, t_1 \rangle$ and $\langle r_{12}, t_2 \rangle$) are *concurrent* if their epochs are executed by different threads.

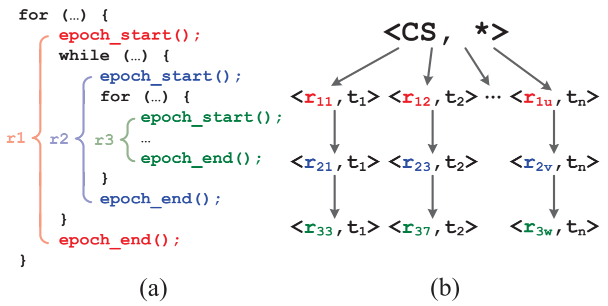

```
for (…) {
  epoch_start();
  while (…) {
    epoch_start();
    for (…) {
      epoch_start();
      …
      epoch_end();
    }
    epoch_end();
  }
  epoch_end();
}
```

Figure 5: An example of regions: (a) a simple program and (b) its region semilattice at some point of execution.

How to Deallocate Regions Correctly and Efficiently? As discussed in §1, a small number of objects may outlive their epochs, and have to be identified and carefully handled during region deallocation. As also discussed in §1, we do not want to solve this problem by an iterative manual process of code refactoring and testing, which is labor-intensive as was done in Facade [50] or Broom [28]. Yak has to automatically accomplish two key tasks: (1) identifying escaping objects and (2) deciding the relocation destination for these objects.

For the first task, Yak uses an efficient algorithm to track cross-region/space references and records all *incoming references* at run time for each region. Right before a region is deallocated, Yak uses these references as the *root set* to compute a transitive closure of objects that can escape the region (details in §5.2).

For the second task, for each escaping object O, Yak tries to relocate O to a live region that will not be deallocated before the last (valid) reference to O. To achieve this goal, Yak identifies the source regions for each incoming cross-region/space reference to O, and *joins* them to find their *least upperbound* on the region semilattice. For example, in Figure 5, *joining* $\langle r_{21}, t_1 \rangle$ and $\langle r_{11}, t_1 \rangle$ returns $\langle r_{11}, t_1 \rangle$, while joining any two concurrent regions returns the CS. Intuitively, if O has references from its parent and grand-parent regions, O should be moved up to its grand-parent. If O has two references coming from regions created by different threads, it has to be moved to the CS.

Upon deallocation, computing a transitive closure of escaping objects while other threads are accessing them may result in an incomplete closure. In addition, moving objects concurrently with other running threads is dangerous and may give rise to data races. Yak employs a lightweight "stop-the-world" treatment to guarantee memory safety in deallocation. When a thread reaches an *epoch_end*, Yak pauses all running threads, scans their stacks, and computes a closure that includes all potential live objects in the deallocating region. These objects are moved to their respective target regions before all mutator threads are resumed.

5 Yak Design and Implementation

We have implemented Yak in Oracle's production JVM OpenJDK 8 (build 25.0-b70). In addition to implementing our own region-based technique, we have modified the two JIT compilers (C1 and Opto), the interpreter, the object/heap layout, and the Parallel Scavenge collector (to manage the CS). Below, we discuss how to split the heap and create regions (§5.1); how to track inter-region/space references, how to identify escaping objects, and how to determine where to move them (§5.2); how to deallocate regions correctly and efficiently (§5.3); and how to modify the Parallel Scavenge GC to collect the CS (§5.4).

5.1 Region & Object Allocation

Region Allocation When the JVM is launched, it asks the OS to reserve a block of virtual addresses based on the maximum heap size specified by the user (*i.e.*, -Xmx). Yak divides this address space into the CS and the DS, with the ratio between them specified by the user via JVM parameters. Yak initially asks the OS to commit a small amount of memory, which will grow if the initial space runs out. Once an *epoch_start* is encountered, Yak creates a region in the DS. A region contains a list of pages whose size can be specified by a JVM parameter.

Heap Layout Figure 6 illustrates the heap layout maintained by Yak. The CS is the same as the old Java heap maintained by a generational GC, except for the newly added *remember set*. The DS is much bigger, containing multiple regions, with each region holding a list of pages.

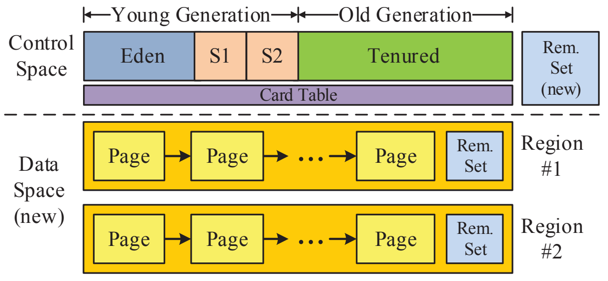


Figure 6: The heap layout in Yak.

The *remember set* is a bookkeeping data structure maintained by Yak for every region and the CS space. It is used to determine what objects escape a region r and where to relocate them. The remember set of CS helps identify live objects in the CS. The remember set of a region/space r is implemented as a hash table that maps an object O in r to all references to O that come from a different region/space.

Note that a remember set is one of the many possible data structures to record such references. For example, the generational GC uses a *card table* that groups objects into fixed-sized buckets and tracks which buckets contain objects with pointers that point to the young generation. Yak uses remember sets, because each region has only a few incoming references; using a card table instead would require us to scan *all objects* from the CS and other regions to find these references.

Allocating Objects in the DS When the execution is in an epoch, we redirect all allocation requests made to the Eden space (*e.g.*, young generation) to our new `Region_Alloc` function. Yak filters out JVM meta-data objects, such as class loader and class objects, from getting allocated in the region. Using a quick *bump pointer* algorithm (which uses a pointer that points to the starting address of free space and bumps it up upon each allocation), the region's manager attempts to allocate the object on the last page of its page list. If this page does not have enough space, the manager creates a new page and appends it to the list. For a large object that cannot fit into one page, we request a special page that can fit the object. For performance, large objects are never moved.

5.2 Tracking Inter-region References

Overview As discussed in §4, Yak needs to efficiently track all inter-region/space references. At a high level, Yak achieves this in three steps. First, Yak adds a 4-byte field *re* into the header space of each object to record the region information of the object. Upon an object allocation, its *re* field is updated to the corresponding region ID. A special ID is used for the CS.

Second, we modify the write barrier (*i.e.*, a piece of code executed with each heap write instruction $a.f = b$) to detect and record heap-based inter-region/space references. Note that, in OpenJDK, a barrier is already required by a generational GC to track inter-generation references. We modify the existing write barrier as shown in Algorithm 1.

Algorithm 1: The write barrier $a.f = b$.

```
Input: Expression $a.f$, Variable $b$
1 if ADDR($O_a$) $\notin$ SPACE(CS) OR ADDR($O_b$) $\notin$ SPACE(CS) then
2     if REGION($O_a$) $\neq$ REGION($O_b$) then
3         Record the reference ADDR($O_a$) + OFFSET($f$) $\xrightarrow{\text{REGION}(O_a)}$ ADDR($O_b$) in the remember set rs of $O_b$'s region
4 ... // Normal OpenJDK logic (for marking the card table)
```

Finally, Yak detects and records local-stack-based inter-region references as well as remote-stack-based references when *epoch_end* is triggered. These algorithms are shown in Lines 1 – 4 and Lines 5 – 10 in Algorithm 2.

Details We describe in detail how Yak can track all inter-region references, following the three places where the reference to an escaping object can reside in – the heap, the local stack, and a remote stack. The semantics of writes to static fields (*i.e.*, globals) as well as array stores are similar to that of instance field accesses; we omit the details of their handling. Copies of large memory regions (*e.g.*, `System.arraycopy`) are also tracked in Yak.

(1) In the heap. An object O_b can outlive its region r if its reference is written into an object O_a allocated in another (live) region r'. Algorithm 1 shows the write barrier to identify such escaping objects O_b. The algorithm checks whether the reference is an inter-region/space reference (Line 2). If it is, the pointee's region (*i.e.*, REGION(O_b)) needs to update its remember set (Line 3).

Each entry in the remember set is a reference which has a form $a \xrightarrow{r} b$ where a and b are the addresses of the pointer and pointee, respectively, and r represents the region the reference comes from. In most cases (such as those represented by Algorithm 1), r is the region in which a resides and it will be used to compute the target region to which b will be moved. However, if a is a stack variable, we need to create a placeholder reference with a special r, determined based on which stack a comes from. We will shortly discuss such cases in Algorithm 2.

To reduce overhead, we have a check that quickly filters out references that do not need to be remembered. As shown in Algorithm 1, if both O_a and O_b are in the same region, including the CS (Lines 1 – 2), we do not need to track that reference, and thus, the barrier proceeds to the normal OpenJDK logic.

(2) On the local stack. An object can escape by being referenced by a stack variable declared beyond the scope of the running epoch. Figure 7 (a) shows a simple example. The reference of the object allocated on Line 3 is assigned to the stack variable a. Because a is still alive after *epoch_end*, it is unsafe to deallocate the object.

Yak identifies this type of escaping objects through an analysis at each *epoch_end* mark. Specifically, Yak scans the local stack of the deallocating thread for the set of live variables at *epoch_end* and checks if an object in r can be referenced by a live variable (Lines 1 – 4 in Algorithm 2). For each such escaping object O_{var}, Yak adds a placeholder incoming reference, whose source is from r's parent region (say p), into the remember set *rs* of r (Line 4). This will cause O_{var} to be relocated to p. If the variable is still live when p is about to be deallocated, this would be detected by the same algorithm and O_{var} would be further relocated to p's parent.

```
1 a = ...;
2 //epoch_start
3 b = new B();
4 if (/*condition */) {
5     a = b;
6 }
7 //epoch_end
8 c = a;
```

(a)

```
1  Thread t:
2  //epoch_start
3  a = A.f;
4  a.g = new O();
5  //epoch_end
6
7  Thread t′:
8  //epoch_start
9  p = A.f;
10 b = p.g;
11 p.g = c;
12 //epoch_end
```

(b)

Figure 7: (a) An object referenced by b escapes its epoch via the stack variable a; (b) An object O created by thread t and referenced by $a.g$ escapes to thread t' via the load statement $b = p.g$.

(3) On the remote stack. A reference to an object O created by thread t could end up in a stack variable in thread t'. For example, in Figure 7 (b), object O created on Line 4 escapes t through the store at the same line and is loaded to the stack of another thread t' on Line 10. A naïve way to track these references is to monitor every read (*i.e.*, a *read barrier*), such as the load on Line 10 in Figure 7 (b).

Yak avoids the need for a read barrier, whose large overhead could affect practicality and performance. Before proceeding to discuss the solution, let us first examine the potential problems of missing a read barrier. The purpose of the read barrier is for us to understand whether a region object is loaded on a remote stack so that the object will not be mistakenly reclaimed when its containing region is deallocated. Without it, a remote thread which references an object O in region r, may cause two potential issues when r is deallocated (Figure 8).

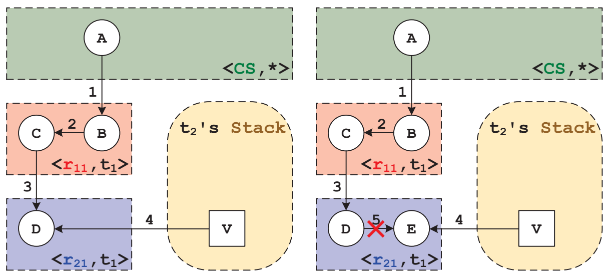


Figure 8: Examples showing potential problems with references on a remote stack: (a) moving object D is dangerous; and (b) object E, which is also live, is missed in the transitive closure.

Problem 1: Dangerous object moving. Figure 8 (a) illustrates this problem. Variable v on the stack of thread t_2 contains a reference to object D in region $\langle r_{21}, t_1\rangle$ (by following the chain of references starting at object A in the CS). When this region is deallocated, D is in the escaping transitive closure; its target region, as determined by the semilattice, is its parent region $\langle r_{11}, t_1\rangle$. Obviously, moving D at the deallocation of $\langle r_{21}, t_1\rangle$ is dangerous, because we are not aware that v references it and thus cannot update v with D's new address after the move.

Problem 2: Dangerous object deallocation. Figure 8 (b) shows this problem. Object E is first referenced by D in the same region $\langle r_{21}, t_1\rangle$. Hence, the remote thread t_2 can reach E by following the reference chain starting at A. Suppose t_2 loads E into a stack variable v and then deletes the reference from D to E. When region $\langle r_{21}, t_1\rangle$ is deallocated, E cannot be included in the escaping transitive closure while it is being accessed by a remote stack. E thus becomes a "dangling" object that would be mistakenly treated as a dead object and reclaimed immediately.

Solution Summary Yak's solution to these problems is to pause all other threads and scan their stacks when thread t deallocates a region r. Objects in r that are also on a remote stack need to be explicitly marked as *escaping roots* before the escaping closure computation because they may be dangling objects (such as E in Figure 8 (b)) that are already disconnected from other objects in the region. §5.3 provides the detailed algorithms for region deallocation and thread stack scanning.

5.3 Region Deallocation

Algorithm 2 shows our region deallocation algorithm that is triggered at each *epoch_end*. This algorithm computes the closure of escaping objects, moves escaping objects to their target regions, and then recycles the whole region.

Algorithm 2: Region deallocation.

```
   Input: Region r, Thread t
1  Map⟨Var, Object⟩ stackObjs ← SCANSTACK(t, r)
2  foreach ⟨var, O_var⟩ ∈ stackObjs do
3      if REGION(O_var) = r then
4          Record a placeholder reference ADDR(var)
           --r.parent--> ADDR(O_var) in r's remember set rs

5  PAUSEOTHERTHREADS()
6  foreach Thread t′ ∈ THREADS() : t′ ≠ t do
7      Map⟨Var, Object⟩ remoteObjs ← SCANSTACK(t′, r)
8      foreach ⟨var, O_var⟩ ∈ remoteObjs do
9          if REGION(O_var) = r then
10             Record a placeholder reference ADDR(var)
               --CS--> ADDR(O_var) in r's remember set rs

11 CLOSURECOMPUTATION()
12 RESUMEPAUSEDTHREADS()
13 Put all pages of r back onto the available page list
```

Finding Escaping Roots There are three kinds of escaping roots for a region r. First, pointees of inter-region/space references recorded in the remember set of r. Second, objects referenced by the local stack of the deallocating thread t. Third, objects referenced by the remote stacks of other threads.

Since inter-region/space references have already been captured by the write barrier (§5.2), here we first identify objects that escape the epoch via t's local stack, as shown in Lines 1 – 4 of Algorithm 2.

Next, Yak identifies objects that escape via remote stacks. To do this, Yak needs to synchronize threads (Line 5). When a remote thread t' is paused, Yak scans its stack variables and returns a set of objects that are referenced by these variables and located in region r. Each such (remotely referenced) object needs to be explicitly marked as an escaping root to be moved to the CS (Line 10) before the transitive closure is computed (Line 11).

No threads are resumed until t completes its closure computation and moves all escaping objects in r to their target regions. Note that it is unsafe to let a remote thread t' proceed even if the stack of t' does not reference any object in r. To illustrate, consider the following scenario. Suppose object A is in the CS and object B is in region r, and there is a reference from A to B. Only A but not B is on the stack of thread t' when r is deallocated. Scanning the stack of t' would not find any new escaping root for r. However, if t' is allowed to proceed immediately, t' could load B onto its stack through A and then delete the reference between A and B. If this occurs before t completes its closure computation, B would not be included in the closure although it is still live.

After all escaping objects are relocated, the entire region is deallocated with all its pages put back onto the free page list (Line 13).

Closure Computation Algorithm 3 shows the details of our closure computation from the set of escaping roots detected above. Since all other threads are paused, closure computation is done together with object moving. The closure is computed based on the remember set *rs* of the current deallocating region r. We first check the remember set *rs* (Line 1): if *rs* is empty, this region contains no escaping objects and hence is safe to be reclaimed. Otherwise, we need to identify all reachable objects and relocate them.

We start off by computing the target region to which each *escaping root* O_b needs to be promoted (Lines 2 – 4). We check each reference $addr \xrightarrow{r'} O_b$ in the remember set and then *join* all the regions r' based on the region semilattice. The results are saved in a map *promote*.

We then iterate through all escaping roots in topological order of their target regions (the loop at Line 5).[2] For each

[2]The order is based on the region semilattice. For example, CS is ordered before any DS region.

Algorithm 3: Closure computation.

```
Input: Remember Set rs of Region r
1  if The remember set rs of r is NOT empty then
2      foreach Escaping root $O_b \in rs$ do
3          foreach Reference $addr \xrightarrow{r'}$ ADDR($O_b$) in rs do
4              $promote[O_b] \leftarrow$ JOIN ($r'$, $promote[O_b]$)
5      foreach Escaping root $O_b$ in topological order of promote[$O_b$] do
6          Region $tgt \leftarrow promote[O_b]$
7          Initialize queue gray with $\{O_b\}$
8          while gray is NOT empty do
9              Object $O \leftarrow$ DEQUEUE(gray)
10             Write tgt into the region field of $O$
11             Object $O^* \leftarrow$ MOVE($O$, tgt)
12             Put a forward reference at ADDR($O$)
13             foreach Reference $addr \xrightarrow{x}$ ADDR($O$) in r's rs do
14                 Write ADDR($O^*$) into addr
15                 if $x \neq tgt$ then
16                     Add reference $addr \xrightarrow{x}$ ADDR($O^*$) into the remember set of region tgt
17             foreach Outgoing reference e of $O^*$ do
18                 Object $O' \leftarrow$ TARGET(e)
19                 if $O'$ is a forward reference then
20                     Write the new address into $O^*$
21                 Region $r' \leftarrow$ REGION($O'$)
22                 if $r' = r$ then
23                     ENQUEUE($O'$, gray)
24                 else if $r' \neq tgt$ then
25                     Add reference ADDR($O^*$) $\xrightarrow{tgt}$ ADDR($O'$) into the remember set of region $r'$
26 Clear the remember set rs of r
```

escaping root O_b, we perform a breadth-first traversal inside the current region to identify a closure of *transitively escaping* objects reachable from O_b and put all of them into a queue *gray*. During this traversal (Lines 8 – 23), we compute the regions to which each (transitively) escaping object should be moved and conduct the move. We will shortly discuss the details.

Identifying Target Regions When a transitively escaping object O' is reachable from only one escaping root O_b, we simply use the target region of O_b as the target of O'. When O' is reachable from multiple escaping roots, which may correspond to different target regions, we use the "highest-ranked" one among them as the target region of O'.

The topological order of our escaping root traversal is key to our implementation of the above idea. By computing closure for a root with a "higher-ranked" region earlier, objects reachable from multiple roots need to be traversed only once – the check at Line 22 filters out those that already have a region r' ($\neq r$) assigned in a previous iteration of the loop because the region to be assigned in the current iteration is guaranteed to be "lower-ranked" than r'. When this case happens, the traversal stops further tracing the outgoing references from O'.

Figure 9 (a) shows a simple heap snapshot when region $\langle r_{21}, t_1 \rangle$ is about to be deallocated. There are two references in its remember set, one from region $\langle r_{11}, t_1 \rangle$ and a second from $\langle r_{12}, t_2 \rangle$. The objects C and D are the escaping roots. Initially, our algorithm determines that C will be moved to $\langle r_{11}, t_1 \rangle$ and D to the CS (because it is reachable from a concurrent region $\langle r_{12}, t_2 \rangle$). Since the CS is higher-ranked than $\langle r_{11}, t_1 \rangle$ in the semilattice, the transitive closure computation for D occurs before C, which sets E's target to the CS. Later, when the transitive closure for C is computed, E will be ignored (since it has been visited).

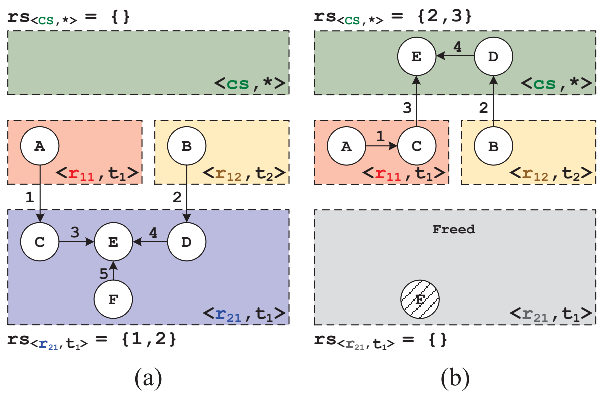


Figure 9: An example heap snapshot (a) before and (b) after the deallocation of region $\langle r_{21}, t_1 \rangle$.

Updating Remember Sets and Moving Objects Because we have pause all threads, object moving is safe (Line 11). When an object O is moved, we need to update *all* (stack and heap) locations that store its references. There can be three kinds of locations from which it is referenced: (1) intra-region locations (*i.e.*, referenced from another object in r); (2) objects from other regions or the CS; and (3) stack locations. We discuss how each of these types is handled by Algorithm 3.

(1) Intra-region locations. To handle intra-region references, we follow the standard GC treatment by putting a special *forward reference* at O's original location (Line 12). This will notify intra-region incoming references of the location change – when this old location of O is reached from another reference, the forward reference there will be used to update the source of that reference (Line 20).

(2) Objects from another region. References from these objects must have been recorded in r's remember set. Hence, we find all inter-region/space references of O in the remember set rs and update the source of each such reference with the new address O^* (Line 14). Since O^* now belongs to a new region tgt, the inter-region/space references that originally went into region r now go into region tgt. If the regions contain such a reference are not tgt, such references need to be explicitly added into the remember set of tgt (Line 16).

When O's outgoing edges are examined, moving O to region tgt may result in new inter-region/space references (Lines 24 – 25). For example, if the target region r' of a pointee object O' is not tgt (*i.e.*, O' has been visited from another escaping root), we need to add a new entry $\text{ADDR}(O^*) \xrightarrow{tgt} \text{ADDR}(O')$ into the remember set of r'.

(3) Stack locations. Since stack locations are also recorded as entries of the remember set, updating them is performed in the same way as updating heap locations. For example, when O is moved, Line 14 would update each reference going to O in the remember set. If O has (local or remote) stack references, they must be in the remember set and updated as well.

After the transitive closure computation and object promotion, the remember set rs of region r is cleared (Line 26).

Figure 9 (b) shows the heap after region $\langle r_{21}, t_1 \rangle$ is deallocated. The objects C, D, and E are escaping objects and will be moved to the target region computed. Since D and E belong to the CS, we add their incoming references 2 and 3 into the remember set of the CS. Object F does not escape the region, and hence, is automatically freed.

5.4 Collecting the CS

We implement two modifications to the Parallel Scavenge GC to collect the CS. First, we make the GC run locally in the CS. If the GC tracing reaches a reference to a region object, we simply ignore the reference.

Second, we include references in the CS' remember set into the tracing roots, so that corresponding CS objects would not be mistakenly reclaimed. Before tracing each such reference, we validate it by comparing the address of its target CS object with the current content in its source location. If they are different, this reference has become invalid and is discarded. Since the Parallel Scavenge GC moves objects (away from the young generation), Yak also needs to update references in the remember set of each region when their source in the CS is moved.

Yak also implements a number of optimizations on the remember set layout, large object allocation, as well as region/thread ID lookup. We omit the details of these optimizations for brevity.

6 Evaluation

This section presents an evaluation of Yak on widely-deployed real-world systems.

6.1 Methodology and Benchmarks

We have evaluated Yak on Hyracks [12], a parallel dataflow engine powering the Apache AsterixDB [2] stack, Hadoop [4], a popular distributed MapReduce [21] implementation, and GraphChi [41], a disk-based graph processing system. These three frameworks were selected due to their popularity and diverse characteristics. For example, Hyracks and Hadoop are distributed frameworks while GraphChi is a single-PC disk-based system. Hyracks runs one JVM on each node with many threads to process data while Hadoop runs multiple JVMs on each node, with each JVM using a small number of threads.

For each framework, we selected a few representative programs, forming a benchmark set with nine programs – external sort (ES), word count (WC), and distributed grep (DG) for Hyracks; in-map combiner (IC), top-word selector (TS), and distributed word filter (DF) for Hadoop; connected components (CC), community detection (CD), and page rank (PR) for GraphChi. Table 1 provides the descriptions of these programs.

FW	P	Description
Hyracks	ES	Sort a large array of data that cannot fit in main memory
	WC	Count word occurrences in a large document
	DG	Find matches based on user-defined regular expressions
Hadoop	IC	Count word frequencies in a corpus using local aggregation
	TS	Select a number of words with most frequent occurrences
	DF	Return text with user-defined words filtered out
GraphChi	PR	Compute page ranks (SpMV kernel)
	CC	Identify strongly connected components (label propagation)
	CD	Detect communities (label propagation)

Table 1: Our benchmarks and their descriptions.

Table 2 shows the datasets and heap configurations in our experiments. For Yak, the heap size is the sum of the sizes of both CS and DS. Since we fed different datasets to various frameworks, their memory requirements were also different. Evidence [11] shows that in general the heap size needs to be at least twice as large as the minimum memory size for the GC to perform well. We selected the heap configurations shown in Table 2 based on this observation – they are roughly 2× – 3× of the minimum heap size needed to run the original JVM.

FW	Dataset	Size	Heap Configs
Hyracks	Yahoo Webmap	72GB	20GB, 24GB
Hadoop	StackOverflow	37GB	2&1GB, 3&2GB

GraphChi	Sample twitter-2010	E = 100M
		V = 62M
			6GB, 8GB

Table 2: Datasets and heap configurations used to run our programs; for Hadoop, the configurations *a*&*b* GB are the max heap sizes for each map (*a*) and reduce task (*b*).

In a small number of cases, the JVM uses hand-crafted assembly code to allocate objects directly into the heap without calling any C/C++ function. While we have spent more than a year on development, we have not yet performed any assembly-based optimizations for Yak. Thus, this assembly-based allocation in the JVM would allow some objects in an epoch to bypass Yak's allocator. To solve the problem, we had to disable this option and force all allocation requests to go through the main allocation entrance in C++. For a fair comparison, we kept the assembly-level allocation option disabled for all experiments including both Yak and original GC runs. We saw a small performance degradation (2–6%) after disabling this option in the JVM.

We ran Hyracks and Hadoop on an 11-node cluster, each with 2 Xeon(R) CPU E5-2640 v3 processors, 32GB memory, 1 SSD, running CentOS 6.6. We ran GraphChi on one node of this cluster, since it is a single-PC system. For Yak, we let the ratio between the sizes of the CS and the DS be 1/10. We did not find this ratio to have much impact on performance as long as the DS is large enough to contain objects created in each epoch. The page size in DS is 32KB by default. We performed experiments with different DS-page sizes and report these results shortly. We focus our comparison between Yak and Parallel Scavenge (PS) – the Oracle JVM's default production GC.

We ran each program for three iterations. The first iteration warmed up the JIT. The performance difference between the last two iterations were negligible (*e.g.*, less than 5%). This section reports the medians. We also confirmed that no incorrect results were produced by Yak.

6.2 Epoch Specification

We performed our annotation by strictly following *existing* framework APIs. For Hyracks, an epoch covers the lifetime of a (user-defined) dataflow operator (*i.e.*, via `open`/`close`); for Hadoop, it includes the body of a Map or Reduce task (*i.e.*, via `setup`/`cleanup`). For GraphChi, we let each epoch contain the body of a sub-interval specified by a `beginSubInterval` callback, since each sub-interval holds and processes many vertices and edges as illustrated in §3. A sub-interval creates many threads to load sliding shards and execute update functions. The body of each such thread is specified as a sub-epoch. It took us about ten minutes to annotate all three programs on each framework. Note that our optimization for these frameworks only scratches the surface; vast opportunities are possible if both user-defined operators and system's built-in operators are epoch-annotated.

6.3 Latency and Throughput

Figure 10 depicts the detailed performance comparisons between Yak and PS. Table 3 summarizes Yak's perfor-

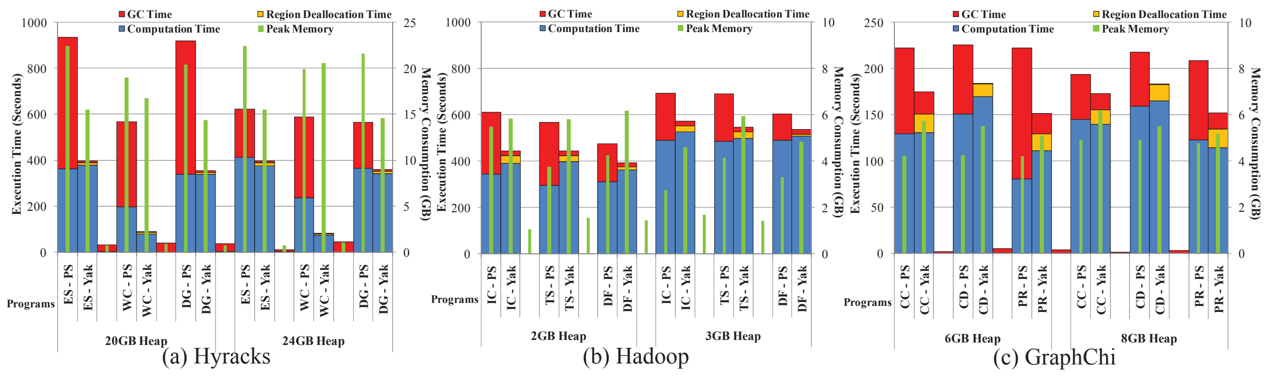


Figure 10: Performance comparisons on various programs; each group compares performance between Parallel Scavenge (PS) and Yak on a program with two "fat" and two "thin" bars. The left and right fat bars show the running times of PS and Yak, respectively, which is further broken down into three components: GC (in red), region deallocation (in orange), application computation (in blue) times. The left and right thin bars show maximum memory consumption of PS and Yak, collected by periodically running `pmap`.

FW	Overall	GC	App	Mem
Hyracks	0.14 ~ 0.64	0.02 ~ 0.11	0.31 ~ 1.05	0.67 ~ 1.03
	(0.40)	(0.05)	(0.77)	(0.78)
Hadoop	0.73 ~ 0.89	0.17 ~ 0.26	1.03 ~ 1.35	1.07 ~ 1.67
	(0.81)	(0.21)	(1.13)	(1.44)
GraphChi	0.70 ~ 0.86	0.15 ~ 0.56	0.91 ~ 1.13	1.07 ~ 1.34
	(0.77)	(0.38)	(1.01)	(1.21)

Table 3: Summary of Yak performance normalized to baseline PS in terms of **Overall** run time, **GC** time, including Yak's region deallocation time, **App**lication (non-GC) time, and **Mem**ory consumption across all settings on each framework. The values shown depict Min ~ Max and (Mean), and are normalized to PS. A lower value indicates better performance versus PS.

mance improvement by showing Overall run time, as well as GC and Application time and Memory consumption, all normalized to those of PS.

For Hyracks, Yak outperforms PS in all evaluated metrics. The GC time is collected by identifying the maximum GC time across runs on all slave nodes. Data-parallel tasks in Hyracks are isolated by design and they do not share any data structures across task instances. Hence, while Yak's write barrier incurs overhead, almost all references captured by the write barrier are intra-region references and thus they do not trigger the slow path of the barrier (*i.e.*, updating the remember set). Yak also improves the (non-GC) application performance — this is because PS only performs thread-local allocation for small objects and the allocation of large objects has to be in the shared heap, protected by locks. In Yak, however, all objects are allocated in thread-local regions and thus threads can allocate objects completely in parallel. Lock-free allocation is the major reason why Yak improves application performance because large objects (*e.g.*, arrays in HashMaps) are frequently allocated in such programs.

For Hadoop and GraphChi, while Yak substantially reduces the GC time and the overall execution time, it increases the application time and memory consumption. Longer application time is expected because (1) memory reclamation (*i.e.*, region deallocation) shifts from the GC to the application execution, with Yak, and (2) the write barrier is triggered to record a large number of references. For example, Hadoop has a state object (*i.e.*, `context`) in the control path that holds objects created in the data path, generating many *inter-space* references. In GraphChi, a number of large data structures are shared among different data-loading threads, leading to many *inter-region* references (*e.g.*, reported in Table 4). Recording all these references makes the barrier overhead stand out.

We envision two approaches that can effectively reduce the write barrier cost. First, existing GCs all have manually crafted/optimized assembly code to implement the write barrier. As mentioned earlier, we have not yet investigated assembly-based optimizations for Yak. We expect the barrier cost to be much lower when these optimizations are implemented. Second, adding extra annotations that define finer-grained epochs may provide further performance improvement. For example, if objects reachable from the state object can be created in the CS in Hadoop, the number of inter-space references can be significantly reduced. In this experiment, we did not perform any program restructuring, but we believe significant performance potential is possible with that: it is up to the developer to decide how much annotation effort she can afford to expend for how much extra performance gain she would like to achieve.

Yak greatly shortens the pauses caused by GC. When Yak is enabled, the maximum (deallocation or GC) pauses in Hyracks, Hadoop, and GraphChi are, respectively, 1.82, 0.55, and 0.72 second(s), while the longest GC pauses under PS are 35.74, 1.24, and 9.48 seconds, respectively.

As the heap size increases, there is a small performance improvement for PS due to fewer GC runs. The heap

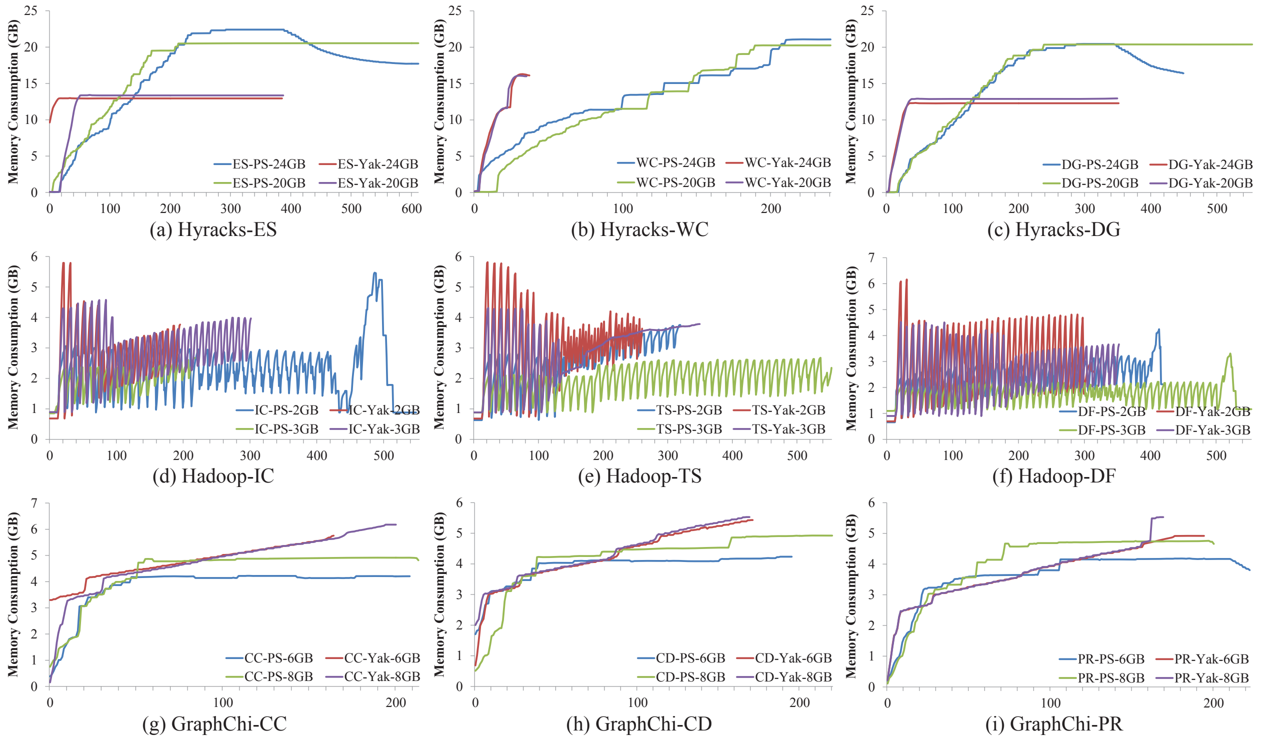


Figure 11: Memory footprints collected from `pmap`.

increase has little impact on Yak's overall performance, given that the CS is small anyways.

6.4 Memory Usage

We measured memory usage by periodically running `pmap` to understand the overall memory consumption of the Java process (for both the application and GC data). Figure 11 compares the memory footprints of Yak and PS under different heap configurations. For Hyracks and GraphChi, memory footprints are generally stable, while Hadoop's memory consumption fluctuates. This is because Hadoop runs multiple JVMs and different JVM instances are frequently created and destroyed. Since the JVM never returns claimed memory back to the OS until it terminates, the memory consumption always grows for Hyracks and GraphChi. The amount of memory consumed by Hadoop, however, drops frequently due to the frequent creation and termination of its JVM processes.

Note that the end times of Yak's memory traces on Hadoop in Figure 11 are earlier than the execution finish time reported in Figure 10. This is because Figure 11 shows the memory trace of the node that has the *highest memory consumption*; the computation on this node often finishes before the entire program finishes.

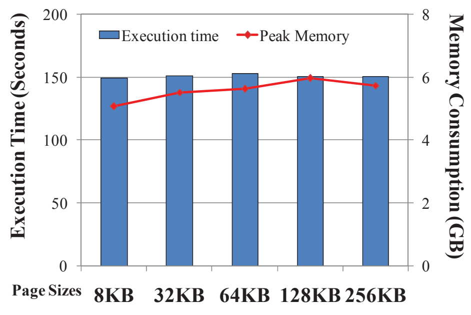
Yak constantly has lower memory consumption than PS for Hyracks. This is primarily because Yak can recycle memory *immediately* when a data processing thread finishes, while there is often a delay before the GC reclaims memory. For Hadoop and GraphChi, Yak has slightly higher memory consumption than PS. The main reason is that there are many control objects created in the data path and allocated in regions. Those objects often have shorter lifespans than their containing regions and, therefore, PS can reclaim them more efficiently than Yak.

Space Overhead To understand the overhead of the extra 4-byte field *re* in each object header, we ran the GraphChi programs with the unmodified HotSpot 1.8.0_74 and compared peak heap consumption with that of Yak (by periodically running `pmap`). We found that the difference (*i.e.*, the overhead) is relatively small. Across the three GraphChi benchmarks, this overhead varies from 1.1% to 20.8%, with an average of 12.2%.

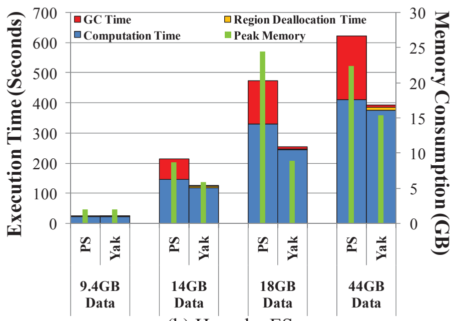
6.5 Performance Breakdown

To provide a deeper understanding of Yak's performance, Table 4 reports various statistics on Yak's heap. Yak was built based on the assumption that in a typical Big Data system, only a small number of objects escape from the data path to the control path. This assumption has been validated by the fact that the ratios between numbers in **#CSR** and **#TR** are generally very small. As a result, each region has only very few objects (**%CSO**) that escape to the CS when the region is deallocated.

Figure 12 (a) depicts execution time and memory performance with Yak, when different page sizes are used. Execution time under different page sizes does not vary



(a) GraphChi-PR-Yak



(b) Hyracks-ES

Figure 12: Performance comparisons between (a) different page sizes when Yak ran on GraphChi PR with a 6GB heap; (b) Yak and PS when datasets of various sizes were sorted by Hyracks ES on a 24GB heap.

Program	#CSR	#CRR	#TR	%CSO	#R
Hyracks-ES	2051	243	3B	0.0028%	103K
Hyracks-WC	2677	4221	213M	0.0043%	148K
Hyracks-DG	2013	16	2B	0.0034%	101K
Hadoop-IC	60K	0	2B	0%	598
Hadoop-TS	60K	0	2B	0%	598
Hadoop-DF	33K	0	1B	0%	598
GraphChi-CC	53K	25K	653M	0.044%	2699
GraphChi-CD	52K	14M	614M	1.3%	2699
GraphChi-PR	54K	24K	548M	0.060%	2699

Table 4: Statistics on Yak's heap: numbers of cross-space references (CSR), cross-region references (CRR), and total references generated by stores (TR); average percentage of objects escaping to the CS (CSO) among all objects in a region when the region retires; and total number of regions created during execution (R).

much (*e.g.*, all times are between 149 and 153 seconds), while the peak memory consumption generally goes up when page size increases (except for the 256KB case).

The write barrier and region deallocation are the two major sources of Yak's application overhead. As shown in Figure 10, region deallocation time accounts for 2.4%-13.1% of total execution time across the benchmarks. Since all of our programs are multi-threaded, it is difficult to pinpoint the exact contribution of the write barrier to execution time. To get an idea of the sensitivity to this barrier's cost, we manually modified GraphChi's execution engine to enforce a barrier between threads that load sliding shards and execute updates. This has the effect of serializing the threads and making the program sequential. For all three programs on GraphChi, we found that the mutator time (*i.e.*, non-pause time) increased by an overall of 24.5%. This shows that the write barrier is a major bottleneck, providing strong motivation for us to hand optimize it in assembly code in the near future.

Scalability To understand how Yak and PS perform when datasets of different sizes are processed, we ran Hyracks ES with four subsets of the Yahoo Webmap with sizes of 9.4GB, 14GB, 18GB, and 44GB respectively. Figure 12 (b) shows that Yak consistently outperforms PS and its performance improvement increases with the size of the dataset processed.

7 Conclusion

We present Yak, a new hybrid Garbage Collector (GC) that can efficiently manage memory in data-intensive applications. Yak treats the data space and control space differently for GC purposes since objects in modern data-processing frameworks follow two vastly-different types of lifetime behavior: data space shows epoch-based object lifetime patterns, whereas the much-smaller control space follows the classic generational lifetime behavior. Yak manages all data-space objects using epoch-based regions and deallocates each region as a whole at the end of an epoch, while efficiently tracking the small number of objects whose lifetimes span region boundaries. Doing so greatly reduces the overheads of traditional generational GC. Our experiments on several real-world applications demonstrate that Yak outperforms the default production GC in OpenJDK on three widely-used real Big Data systems, requiring almost zero user effort.

Acknowledgments

We would like to thank the many OSDI reviewers for their valuable and thorough comments. We are especially grateful to our shepherd Dushyanth Narayanan for his tireless effort to read many versions of the paper and provide suggestions, helping us improve the paper substantially. We thank Kathryn S. McKinley for her help with the preparation of the final version. We also appreciate the feedback from the MIT PDOS group (especially Tej Chajed for sending us the feedback).

This work is supported by NSF grants CCF-0846195, CCF-1217854, CNS-1228995, CCF-1319786, CNS-1321179, CCF-1409423, CCF-1409829, CCF-1439091, CCF-1514189, CNS-1514256, IIS-1546543, CNS-1613023, by ONR grants N00014-16-1-2149 and N00014-16-1-2913, and by a Sloan Fellowship.

References

[1] Aiken, A., Fähndrich, M., and Levien, R. Better static memory management: improving region-based analysis of higher-order languages. In *PLDI* (1995), pp. 174–185.

[2] Alsubaiee, S., Altowim, Y., Altwaijry, H., Behm, A., Borkar, V. R., Bu, Y., Carey, M. J., Cetindil, I., Cheelangi, M., Faraaz, K., Gabrielova, E., Grover, R., Heilbron, Z., Kim, Y., Li, C., Li, G., Ok, J. M., Onose, N., Pirzadeh, P., Tsotras, V. J., Vernica, R., Wen, J., and Westmann, T. AsterixDB: A scalable, open source BDMS. *Proc. VLDB Endow. 7*, 14 (2014), 1905–1916.

[3] Giraph: Open-source implementation of Pregel. http://incubator.apache.org/giraph/.

[4] Hadoop: Open-source implementation of MapReduce. http://hadoop.apache.org.

[5] Appel, A. W. Simple generational garbage collection and fast allocation. *Softw. Pract. Exper. 19*, 2 (1989), 171–183.

[6] Baker, Jr., H. G. List processing in real time on a serial computer. *Commun. ACM 21*, 4 (1978), 280–294.

[7] BEA Systems Inc. Using the Jrockit runtime analyzer. http://edocs.bea.com/wljrockit/docs142/usingJRA/looking.html, 2007.

[8] Beebee, W. S., and Rinard, M. C. An implementation of scoped memory for real-time Java. In *EMSOFT* (2001), pp. 289–305.

[9] Blackburn, S. M., Garner, R., Hoffman, C., Khan, A. M., McKinley, K. S., Bentzur, R., Diwan, A., Feinberg, D., Frampton, D., Guyer, S. Z., Hirzel, M., Hosking, A., Jump, M., Lee, H., Moss, J. E. B., Phansalkar, A., Stefanović, D., VanDrunen, T., von Dincklage, D., and Wiedermann, B. The DaCapo benchmarks: Java benchmarking development and analysis. In *OOPSLA* (2006), pp. 169–190.

[10] Blackburn, S. M., Jones, R., McKinley, K. S., and Moss, J. E. B. Beltway: Getting around garbage collection gridlock. In *PLDI* (2002), pp. 153–164.

[11] Blackburn, S. M., and McKinley, K. S. Immix: a mark-region garbage collector with space efficiency, fast collection, and mutator performance. In *PLDI* (2008), pp. 22–32.

[12] Borkar, V. R., Carey, M. J., Grover, R., Onose, N., and Vernica, R. Hyracks: A flexible and extensible foundation for data-intensive computing. In *ICDE* (2011), pp. 1151–1162.

[13] Borman, S. Sensible sanitation understanding the IBM Java garbage collector. http://www.ibm.com/developerworks/ibm/library/i-garbage1/, 2002.

[14] Boyapati, C., Salcianu, A., Beebee, Jr., W., and Rinard, M. Ownership types for safe region-based memory management in real-time Java. In *PLDI* (2003), pp. 324–337.

[15] Bu, Y., Borkar, V., Xu, G., and Carey, M. J. A bloat-aware design for big data applications. In *ISMM* (2013), pp. 119–130.

[16] Chaiken, R., Jenkins, B., Larson, P.-A., Ramsey, B., Shakib, D., Weaver, S., and Zhou, J. SCOPE: easy and efficient parallel processing of massive data sets. *Proc. VLDB Endow. 1*, 2 (2008), 1265–1276.

[17] Cheney, C. J. A nonrecursive list compacting algorithm. *Commun. ACM 13*, 11 (1970), 677–678.

[18] Cherem, S., and Rugina, R. Region analysis and transformation for Java programs. In *ISMM* (2004), pp. 85–96.

[19] Cohen, J., and Nicolau, A. Comparison of compacting algorithms for garbage collection. *ACM Trans. Program. Lang. Syst. 5*, 4 (1983), 532–553.

[20] Condie, T., Conway, N., Alvaro, P., Hellerstein, J. M., Elmeleegy, K., and Sears, R. MapReduce online. In *NSDI* (2010), pp. 21–21.

[21] Dean, J., and Ghemawat, S. MapReduce: Simplified data processing on large clusters. In *OSDI* (2004), pp. 137–150.

[22] Detlefs, D., Flood, C., Heller, S., and Printezis, T. Garbage-first garbage collection. In *ISMM* (2004), pp. 37–48.

[23] Fang, L., Nguyen, K., Xu, G., Demsky, B., and Lu, S. Interruptible tasks: Treating memory pressure as interrupts for highly scalable data-parallel programs. In *SOSP* (2015), pp. 394–409.

[24] Feng, Y., and Berger, E. D. A locality-improving dynamic memory allocator. In *MSP* (2005), pp. 68–77.

[25] Gay, D., and Aiken, A. Memory management with explicit regions. In *PLDI* (1998), pp. 313–323.

[26] GAY, D., AND AIKEN, A. Language support for regions. In *PLDI* (2001), pp. 70–80.

[27] GIDRA, L., THOMAS, G., SOPENA, J., SHAPIRO, M., AND NGUYEN, N. NumaGiC: A garbage collector for big data on big NUMA machines. In *ASPLOS* (2015), pp. 661–673.

[28] GOG, I., GICEVA, J., SCHWARZKOPF, M., VASWANI, K., VYTINIOTIS, D., RAMALINGAM, G., COSTA, M., MURRAY, D. G., HAND, S., AND ISARD, M. Broom: Sweeping out garbage collection from big data systems. In *HotOS* (2015).

[29] GROSSMAN, D., MORRISETT, G., JIM, T., HICKS, M., WANG, Y., AND CHENEY, J. Region-based memory management in Cyclone. In *PLDI* (2002), pp. 282–293.

[30] HALLENBERG, N., ELSMAN, M., AND TOFTE, M. Combining region inference and garbage collection. In *PLDI* (2002), pp. 141–152.

[31] HARRIS, T. Early storage reclamation in a tracing garbage collector. *SIGPLAN Not. 34*, 4 (Apr. 1999), 46–53.

[32] HICKS, M., MORRISETT, G., GROSSMAN, D., AND JIM, T. Experience with safe manual memory-management in Cyclone. In *ISMM* (2004), pp. 73–84.

[33] HIRZEL, M., DIWAN, A., AND HERTZ, M. Connectivity-based garbage collection. In *OOPSLA* (2003), pp. 359–373.

[34] HUDSON, R. L., AND MOSS, J. E. B. Incremental collection of mature objects. In *IWMM* (1992), pp. 388–403.

[35] ISARD, M., BUDIU, M., YU, Y., BIRRELL, A., AND FETTERLY, D. Dryad: distributed data-parallel programs from sequential building blocks. In *EuroSys* (2007), pp. 59–72.

[36] JOAO, J. A., MUTLU, O., KIM, H., AGARWAL, R., AND PATT, Y. N. Improving the performance of object-oriented languages with dynamic predication of indirect jumps. In *ASPLOS* (2008), pp. 80–90.

[37] JOAO, J. A., MUTLU, O., AND PATT, Y. N. Flexible reference counting-based hardware acceleration for garbage collection. In *ISCA* (2009), pp. 418–428.

[38] KERMANY, H., AND PETRANK, E. The Compressor: Concurrent, incremental, and parallel compaction. In *PLDI* (2006), pp. 354–363.

[39] KIM, H., JOAO, J. A., MUTLU, O., LEE, C. J., PATT, Y. N., AND COHN, R. VPC prediction: Reducing the cost of indirect branches via hardware-based dynamic devirtualization. In *ISCA* (2007), pp. 424–435.

[40] KOWSHIK, S., DHURJATI, D., AND ADVE, V. Ensuring code safety without runtime checks for real-time control systems. In *CASES* (2002), pp. 288–297.

[41] KYROLA, A., BLELLOCH, G., AND GUESTRIN, C. GraphChi: Large-Scale Graph Computation on Just a PC. In *OSDI* (2012), pp. 31–46.

[42] LU, L., SHI, X., ZHOU, Y., ZHANG, X., JIN, H., PEI, C., HE, L., AND GENG, Y. Lifetime-based memory management for distributed data processing systems. *Proc. VLDB Endow. 9*, 12 (2016), 936–947.

[43] MAAS, M., HARRIS, T., ASANOVIĆ, K., AND KUBIATOWICZ, J. Trash Day: Coordinating garbage collection in distributed systems. In *HotOS* (2015).

[44] MAAS, M., HARRIS, T., ASANOVIĆ, K., AND KUBIATOWICZ, J. Taurus: A holistic language runtime system for coordinating distributed managed-language applications. In *ASPLOS* (2016), pp. 457–471.

[45] MAKHOLM, H. A region-based memory manager for Prolog. In *ISMM* (2000), pp. 25–34.

[46] MCCARTHY, J. Recursive functions of symbolic expressions and their computation by machine, part i. *Commun. ACM 3*, 4 (Apr. 1960), 184–195.

[47] MITCHELL, N., AND SEVITSKY, G. The causes of bloat, the limits of health. In *OOPSLA* (2007), pp. 245–260.

[48] MURRAY, D. G., MCSHERRY, F., ISAACS, R., ISARD, M., BARHAM, P., AND ABADI, M. Naiad: A timely dataflow system. In *SOSP* (2013), pp. 439–455.

[49] NGUYEN, K., FANG, L., XU, G., AND DEMSKY, B. Speculative region-based memory management for big data systems. In *PLOS* (2015), pp. 27–32.

[50] NGUYEN, K., WANG, K., BU, Y., FANG, L., HU, J., AND XU, G. FACADE: A compiler and runtime for (almost) object-bounded big data applications. In *ASPLOS* (2015), pp. 675–690.

[51] NGUYEN, K., AND XU, G. Cachetor: detecting cacheable data to remove bloat. In *FSE* (2013), pp. 268–278.

[52] OLSTON, C., REED, B., SRIVASTAVA, U., KUMAR, R., AND TOMKINS, A. Pig Latin: a not-so-foreign language for data processing. In *SIGMOD* (2008), pp. 1099–1110.

[53] PIKE, R., DORWARD, S., GRIESEMER, R., AND QUINLAN, S. Interpreting the data: Parallel analysis with Sawzall. *Sci. Program. 13*, 4 (2005), 277–298.

[54] QIAN, F., AND HENDREN, L. An adaptive, region-based allocator for Java. In *ISMM* (2002), pp. 127–138.

[55] SACHINDRAN, N., MOSS, J. E. B., AND BERGER, E. D. Mc^2: High-performance garbage collection for memory-constrained environments. In *OOPSLA* (2004), pp. 81–98.

[56] STEFANOVIĆ, D., MCKINLEY, K. S., AND MOSS, J. E. B. Age-based garbage collection. In *OOPSLA* (1999), pp. 370–381.

[57] THUSOO, A., SARMA, J. S., JAIN, N., SHAO, Z., CHAKKA, P., ANTHONY, S., LIU, H., WYCKOFF, P., AND MURTHY, R. Hive: a warehousing solution over a map-reduce framework. *Proc. VLDB Endow. 2*, 2 (2009), 1626–1629.

[58] TOFTE, M., AND TALPIN, J.-P. Implementation of the typed call-by-value lamda-calculus using a stack of regions. In *POPL* (1994), pp. 188–201.

[59] XU, G. Finding reusable data structures. In *OOPSLA* (2012), pp. 1017–1034.

[60] XU, G., ARNOLD, M., MITCHELL, N., ROUNTEV, A., SCHONBERG, E., AND SEVITSKY, G. Finding low-utility data structures. In *PLDI* (2010), pp. 174–186.

[61] XU, G., ARNOLD, M., MITCHELL, N., ROUNTEV, A., AND SEVITSKY, G. Go with the flow: Profiling copies to find runtime bloat. In *PLDI* (2009), pp. 419–430.

[62] XU, G., MITCHELL, N., ARNOLD, M., ROUNTEV, A., AND SEVITSKY, G. Software bloat analysis: Finding, removing, and preventing performance problems in modern large-scale object-oriented applications. In *FoSER* (2010), pp. 421–426.

[63] XU, G., AND ROUNTEV, A. Detecting inefficiently-used containers to avoid bloat. In *PLDI* (2010), pp. 160–173.

[64] YANG, H.-C., DASDAN, A., HSIAO, R.-L., AND PARKER, D. S. Map-reduce-merge: simplified relational data processing on large clusters. In *SIGMOD* (2007), pp. 1029–1040.

[65] YU, Y., GUNDA, P. K., AND ISARD, M. Distributed aggregation for data-parallel computing: Interfaces and implementations. In *SOSP* (2009), pp. 247–260.

[66] YU, Y., ISARD, M., FETTERLY, D., BUDIU, M., ERLINGSSON, U., GUNDA, P. K., AND CURREY, J. DryadLINQ: a system for general-purpose distributed data-parallel computing using a high-level language. In *OSDI* (2008), pp. 1–14.

[67] ZAHARIA, M., CHOWDHURY, M., FRANKLIN, M. J., SHENKER, S., AND STOICA, I. Spark: Cluster computing with working sets. HotCloud, p. 10.

Shuffler: Fast and Deployable Continuous Code Re-Randomization

David Williams-King[1] Graham Gobieski[1] Kent Williams-King[2]
James P. Blake[1] Xinhao Yuan[1] Patrick Colp[2] Michelle Zheng[1]
Vasileios P. Kemerlis[3] Junfeng Yang[1] William Aiello[2]

[1]Columbia University [2]University of British Columbia [3]Brown University

Abstract

While code injection attacks have been virtually eliminated on modern systems, programs today remain vulnerable to code reuse attacks. Particularly pernicious are Just-In-Time ROP (JIT-ROP) techniques, where an attacker uses a memory disclosure vulnerability to discover code gadgets at runtime. We designed a code-reuse defense, called *Shuffler*, which continuously re-randomizes code locations on the order of milliseconds, introducing a real-time deadline on the attacker. This deadline makes it extremely difficult to form a complete exploit, particularly against server programs that often sit tens of milliseconds away from attacker machines.

Shuffler focuses on being fast, self-hosting, and nonintrusive to the end user. Specifically, for speed, Shuffler randomizes code asynchronously in a separate thread and atomically switches from one code copy to the next. For security, Shuffler adopts an "egalitarian" principle and randomizes itself the same way it does the target. Lastly, to deploy Shuffler, no source, kernel, compiler, or hardware modifications are necessary.

Evaluation shows that Shuffler defends against all known forms of code reuse, including ROP, direct JIT-ROP, indirect JIT-ROP, and Blind ROP. We observed 14.9% overhead on SPEC CPU when shuffling every 50 ms, and ran Shuffler on real-world applications such as Nginx. We showed that the shuffled Nginx scales up to 24 worker processes on 12 cores.

1 Introduction

At present, programs hardened with the latest mainline protection mechanisms remain vulnerable to code reuse attacks. In a typical scenario, the attacker seizes control of the instruction pointer and executes a sequence of existing code fragments to form an exploit [54]. This is fundamentally very difficult to defend against, as the program must be able to run its own code, and yet the attacker should be prevented from running out-of-order instruction sequences of that same code. One popular mitigation is to deny the attacker knowledge about the program's code through randomization. Unfortunately, memory disclosure vulnerabilities are common in the real world, with 500–2000 discovered per year over the last three years [20]. Such vulnerabilities can be used to read the program's code, at runtime, and unravel any static randomization in a so-called Just-In-Time ROP (JIT-ROP) attack [55].

We propose a system, called *Shuffler*, which provides a deployable defense against JIT-ROP and other code reuse attacks. Other such defenses have appeared in the literature, but all have had significant barriers to deployment: some utilize a custom hypervisor [4, 17, 33, 57]; others involve a modified compiler [7,10,13,40,42], runtime [10,42], or operating system kernel [4,7,17]. Note that there is a security risk in any solution that requires additional privileges, as an attacker can potentially gain access to that elevated privilege level. Also, modified components present a large barrier to the adoption of the system and have less chance of incorporating upstream patches and updates, so users may continue to run vulnerable software versions. In comparison, Shuffler runs in userspace alongside the target program, and requires no system modifications beyond a minimal patch to the loader. Shuffler can be deployed amongst existing cloud infrastructure, adopted by software distributors, or used at small scale by individual security-conscious users.

Shuffler operates by performing continuous code re-randomization at runtime, within the same address space as the programs it defends. Most defenses operating at the same level of privilege as their target do not consider defending their own attack surface. In contrast, we bootstrap into a self-hosted and self-modifying *egalitarian* environment—Shuffler actually shuffles itself. We also defend all of a program's shared libraries, and handle multithreading and process `forks`, shuffling each child independently. Our current prototype does not handle certain hand-coded assembly, but in principle, *all* executable code in a process's address space can be shuffled.

With Shuffler, we aim to rapidly obsolete leaked information by rearranging memory as fast as possible. Shuffler operates within a real-time deadline, which we call the *shuffle period*. This deadline constrains the total execution time available to any attack, since no information about the memory layout transfers from one shuffle period to the next. We achieve a shuffle period on the order of tens of milliseconds, so fast that it is nearly impossible to form a complete exploit. Shuffler creates new function permutations asynchronously in a separate thread, and then atomically migrates program execution from one copy of code to the next. This migration requires a vanishingly small global pause time, as program threads continue to execute unhindered 99.7% of the time (according to SPEC CPU experiments). Thus, if the host machine has a spare CPU core, shuffling at faster rates does not significantly impact the target's performance. Shuffler's default behaviour is to use a fixed shuffling rate, but it can work with different policies. For instance, if the system is under reduced load, a new vulnerability is announced, or an intrusion detection system raises an alarm, the shuffling rate can be increased dynamically.

Our system operates on program binaries, analyzing them and performing binary rewriting. This analysis must be complete and precise; missing even a single code pointer and failing to update it upon re-randomization can cause correctness issues. Because of the difficulty of binary analysis, we leverage existing compiler and linker flags to preserve symbols and relocations. Some (but not all [46]) vendors strip symbol information from binaries to impede reverse engineering, but reversing stripped binaries is still feasible using disassemblers like IDA Pro [27]. We anticipate that vendors would be willing to include (obfuscated) symbols and relocations in their binaries, given the additional defensive possibilities. For instance, relocations enable shuffling but are also required for executable base address randomization on Windows. In the open-source Linux world, high-level build systems are already designed to support the introduction of additional compiler flags [26], allowing distribution-wide security hardening [25, 29, 58].

Evaluation shows that our system successfully defends against all known forms of code reuse, including ROP, direct JIT-ROP, indirect JIT-ROP, and Blind ROP. We ran Shuffler on a range of programs including web servers, databases, and Mozilla's SpiderMonkey Javascript interpreter. We successfully defend against a Blind ROP attack on Nginx, and against a JIT-ROP attack on a toy web server. Shuffler incurs 14.9% overhead on SPEC CPU when shuffling every 50 ms, and has good scalability on Nginx when shuffling up to 24 workers every 50 ms. We show that a 50 ms shuffle period is orders of magnitude faster than the time required by existing JIT-ROP attacks, which take 2.3 to 378 seconds to complete [52, 55].

Our main contributions are as follows:

1. **Deployability:** We design a re-randomization defense against JIT-ROP and code reuse, which runs without modification to the source, compiler, linker, or kernel, and with minimal changes to the loader.

2. **Speed:** We introduce a real-time deadline on the order of milliseconds for any disclosure-based attack, using a new asynchronous re-randomization architecture that has low latency and low overhead.

3. **Egalitarianism:** We describe how we bootstrap our defense into a self-hosting environment, thus avoiding any expansion of the trusted computing base.

4. **Augmented binary analysis:** We show that complete and precise analysis is possible on binaries by leveraging information available from today's compilers (namely, symbols and relocations).

2 Background and Threat Model

Attack taxonomy Many attacks seen in the wild against running programs are based on control-flow hijacking. An attacker uses a memory corruption vulnerability to overwrite control data, like return addresses or function pointers, and branches to a location of their choosing [2]. In the early days, that location could be a buffer where the attacker had directly written their desired exploit code, thus enacting a so-called *code injection* attack. Nowadays, the widespread deployment of Write-XOR-Execute (W^X) [15] ensures that pages cannot be both executable and writable, which has led to the effective demise of code injection.

In response, attackers began to create *code reuse* attacks, stitching together pieces of code already present in a program's code section. The first and simplest such attack was return-to-libc (`ret2libc`) [51, 56], where an attacker redirects control flow to reuse whole `libc` functions, such as `system`, after setting up arguments on the stack. A more sophisticated technique called Return-Oriented Programming (ROP) [54] was soon discovered, where an attacker stitches together very short instruction sequences ending with a return instruction (or other indirect branch instructions [9, 36])—sequences known as gadgets. The terminating return instruction allows the attacker to jump to the next gadget, and the attacker may set up the stack to contain the addresses of a desired "chain" of gadgets. ROP has been shown to be Turing-complete, and there are tools known as ROP compilers which can automatically generate ROP chains [52].

Defenses against code reuse The research community has proposed two main categories of defenses against code reuse. The first is Control Flow Integrity (CFI) [1], which tries to ensure that every indirect branch taken by the program is in accordance with its control-flow graph. However, both coarse-grained CFI [61, 62] and fine-grained CFI [47] can be bypassed through careful selection of gadgets [11, 23, 28].

The second category of defense is code randomization, performed at load-time to make the addresses of gadgets unpredictable. Module-level Address Space Layout Randomization (ASLR) is currently deployed in all major operating systems [49, 60]. Fine-grained randomization schemes have been proposed at the function [6], basic block [59], and instruction [38] level. These defenses spurred a noteworthy new attack called Just-In-Time ROP (JIT-ROP) in 2013 [55]. In JIT-ROP, the attacker starts with one known code address, recursively reads code pages at runtime with a memory disclosure vulnerability, then compiles an attack using gadgets in the exfiltrated code. The authors conclude that no load-time randomization scheme can stand against this attack.

Defenses in the JIT-ROP era The first defenses against JIT-ROP concentrated on preventing recursive gadget harvesting. Oxymoron [5] and Code Pointer Integrity [40] proposed an inaccessible table to hide the true destination of `call` instructions. Other works proposed execute-only memory, either with a custom hypervisor [17, 57] or software emulation [4, 33]. Unfortunately, preventing the direct disclosure of memory pages is insufficient. Indirect JIT-ROP [14, 24] shows that harvesting code pointers from data pages allows the location of gadgets to be inferred, without ever being read. Leakage-resilient diversification [10, 17] combines execute-only memory with fine-grained ASLR and function trampolines. Thus, code pages cannot be read and their contents cannot be inferred through pointers. This defense is currently still effective, though implementing execute-only memory without extensive system modifications remains challenging.

Continuous re-randomization Following a handful of early re-randomization schemes [19, 34], researchers began to realize that continuous re-randomization can defend against JIT-ROP. If code is re-randomized between the time it is leaked and when a gadget chain is invoked, the attack will fail because the gadgets no longer exist.

For instance, Remix [13] continuously re-randomizes the basic block ordering within functions, so that gadgets no longer stay at constant offsets. The system utilizes an LLVM compiler pass to add padding NOPs so that there will be enough space to reorder blocks. However, this intra-function randomization is vulnerable to attacks that leverage function locations or reuse function pointers.

The closest system to Shuffler is TASR [7]. TASR is a source-level technique which performs re-randomization based on pairs of read/write system calls, between any program output (which may leak information) and any program input (which may contain an exploit). However, TASR requires kernel and compiler modifications, is currently only applicable to C programs, and has high performance overhead, as we discuss in Section 5.5.

Finally, another form of ROP called Blind ROP [8] targets servers that fork workers. Since the workers inherit the parent's address space layout, Blind ROP brute forces them without worrying about causing crashes. RuntimeASLR [42] uses heavyweight instrumentation to allow re-randomization of the child process on fork.

2.1 Threat Model

Shuffler is built upon continuous re-randomization. We aim to defend against all known forms of code reuse attacks, including ROP, direct JIT-ROP, indirect JIT-ROP, and Blind ROP. We assume that protection against code injection (W^X) is in place, and that an x86-64 architecture is in use. Our system does not require (and, in fact, is orthogonal to) other defensive techniques like intra-function ASLR, stack smashing protection, or any other compiler hardening technique.

On the attacker's side, we assume:

1. The attacker is performing a code reuse attack, and not code injection (handled by W^X [15]) or a data-only attack [12] (outside the scope of Shuffler).

2. The attacker has access to 1) a memory disclosure vulnerability that may be invoked repeatedly to read arbitrary memory locations, and 2) a memory corruption vulnerability for bootstrapping exploits.

3. Any memory read or write that violates memory permissions (or targets an unmapped page) will cause a detectable crash, and the attacker has no meta-information about page mappings.[1]

4. The attacker knows the re-randomization rate and can time their attack to start at the very beginning of a shuffling period, maximizing the time that code addresses remain the same.

Our technique is particularly effective when defending long-lived processes and network-facing applications, such as servers. Note that network-based attackers have additional latency induced by communication delays, each time they invoke a vulnerability; see Section 6.3 for details.

3 Design

This section presents the design goals of Shuffler, along with its architecture, and outlines significant technical challenges.

[1]Such as access to `/proc/<pid>/maps`.

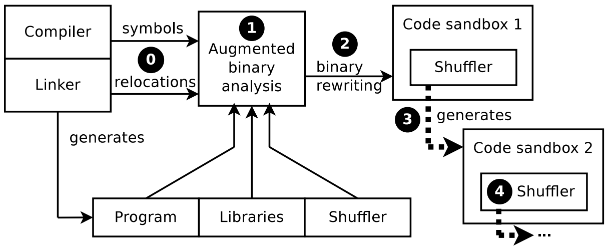


Figure 1: Shuffler architecture. We use symbols and relocations (0) for augmented binary analysis (1), rewrite code into shufflable form (2), and asynchronously create new code copies at runtime (3), while self-hosting (4).

3.1 Goals

The main goals of Shuffler are:

- **Deployability:** We aim to reduce the burden on end-users as much as possible. Thus, we require no direct access to source code, no static binary rewriting on disk, and no modifications to system components (except our small loader patch).
- **Security:** Our goal is to defeat all known code reuse attacks, without expanding the trusted computing base. We constrain the lifetime of leaked information by providing a configurable shuffling period, mitigating code reuse and JIT-ROP attacks.
- **Performance:** Because time is an integral part of our security model, speed is of the essence. We aim to provide low runtime overhead, and also low total shuffling latency to allow for high shuffling rates.

3.2 Architecture

Shuffler is designed to require minimal system modifications. To avoid kernel changes, it runs entirely in userspace; to avoid requiring source or a modified compiler, it operates on program binaries. Performing re-randomization soundly requires complete and precise pointer analysis. Rather than attempting arbitrary binary analysis, we leverage symbol and relocation information from the (unmodified) compiler and linker. Options to preserve this information exist in every major compiler. Thus, we are able to achieve completely accurate disassembly in what we call *augmented binary analysis*—as shown in Figure 1 part (1) and detailed in Section 3.3.

At load-time, Shuffler transforms the program's code using binary rewriting (Figure 1 part (2)). The goal of rewriting is to be able to track and update all code pointers at runtime. We avoid the taint tracking used by related work [7,42] because it is expensive and would introduce races during asynchronous pointer updates. Instead, we leverage our complete and accurate disassembly to transform all code pointers into unique identifiers—indices into a *code pointer table*. These indices cannot be altered after load time (the potential security implications of this choice are discussed in Section 6), but they trade off very favorably against performance and ease of implementation. We handle return addresses (dynamically generated code pointers) differently, encrypting them on the stack rather than using indices, thereby preventing disclosure while maintaining good performance.

Our system performs re-randomization at the level of functions within a specific *shuffle period*, a randomization deadline specified in milliseconds. Shuffler runs in a separate thread and prepares a new shuffled copy of code within this deadline, as shown in Figure 1 part (3). This step is accelerated using a Fenwick tree (see Section 4.4). The vast majority of the re-randomization process is performed asynchronously: creating new copies of code, fixing up instruction displacements, updating pointers in the code table, *etc.* The threads are globally paused only to atomically update return addresses. Since any existing return addresses reference the old copy of code, we must revisit saved stack frames and update them. Each thread walks its own stack in parallel, following base pointers backwards to iterate through stack frames (a process known as *stack unwinding*); see Section 3.3 for details.

Shuffler runs in an *egalitarian* manner, at the same level of privilege as target programs, and within the same address space. To prevent our own code from being used in a code reuse attack, Shuffler randomizes it the same way it does all other code (Figure 1 part (4)). In fact, our scheme uses binary rewriting to transform all code in a userspace application (the program, Shuffler, and all shared libraries) into a single code sandbox, essentially turning it into a statically linked application at runtime. Bootstrapping from original code into this self-hosting environment is challenging, particularly without substantially changing the system loader.

3.3 Challenges

Changing function pointer behaviour Normal binary code is generated under the assumption that the program's memory layout remains consistent and function pointers have indefinite lifetime. Re-randomization introduces an arbitrary lifetime for each block of code, and so re-randomization becomes an exercise in avoiding dangling code pointers. Failing to update even one such pointer may cause the program to crash, or worse, fall victim to a use-after-free attack.

Hence, we need to accurately track and update every code pointer during the re-randomization process. We opt to statically transform all code pointers into unique identifiers—namely, indices into a hidden *code pointer table*. Relying on accurate and complete disassembly (discussed next), we transform all initialization points to use indices. Then, wherever the code pointer is copied throughout memory, it will continue to refer to the same entry in the table. This scheme does not affect the semantics of function pointer comparison. Iterating through and updating the pointer values stored in the table can be done quickly and asynchronously.

Some code pointers are dynamically generated, in particular, return addresses on the stack. We could dynamically allocate table indices, but on the x86 architecture, `call`/`ret` pairs are highly optimized, and replacing them with the table mechanism would involve a large performance degradation [22, 43]. Instead, we allow ordinary calls to proceed as usual, and at re-randomization time we unwind the stack and update return addresses to new values. Rather than leave return addresses exposed on the stack, we encrypt each address with an XOR cipher. Every callee is responsible for disguising the return address on the top of the stack, encrypting it at function entry and decrypting before any function exit. Callers, meanwhile, are responsible for erasing the (now unencrypted) return address immediately after the called function returns. Even though the address is never used by the program, it is still a (leakable) dangling reference. The encryption key can be unique to each function and changed during each stack unwind; see Section 4.1.

Augmented binary analysis The commonly accepted wisdom is that program analysis can be performed at the source level (requiring access to source code) or at the binary level (plagued with completeness issues). In this work, we propose a middle ground, *augmented binary analysis*, which involves analyzing program binaries that have additional information included by the compiler. Compiler-generated binaries are much more amenable to analysis than hand-crafted binaries. We use existing compiler flags and have no visibility into the source code, and yet can achieve complete disassembly.

The common problems with binary analysis are distinguishing code from data, and distinguishing pointers from integers. To tackle these issues, we require that (a) the compiler preserve the symbol table, and (b) that the linker preserve relocations. The symbol table indicates all valid `call` targets and makes disassembly straightforward—we iterate through symbols and disassemble each one independently; there is no need for a linear sweep or recursive traversal algorithm [53]. Relocations are used to indicate portions of an object file (or executable) that need to be patched up once its base address is known. Since each base address is initially zero, every absolute code pointer must have a relocation—but as object files are linked together, most code pointers get resolved and their relocations are discarded. We simply ask the linker to preserve these relocations.

These two augmentations enable complete and accurate disassembly, for any optimization level—at least on the $\sim$30 programs that we tested, many of which have sizable codebases. We describe the details of our augmented binary analysis in Section 4.2.

Bootstrapping into shuffled code As stated above, Shuffler defends its own code the same way it defends all other code—leading to a difficult bootstrapping problem. Shuffled code cannot start running until the code pointer table is initialized, requiring some unshuffled startup code. Shuffled and original code are incompatible if they use code pointers; the process of transforming code pointers to indices overwrites data that the original code accesses, and then the original code will no longer execute correctly. For example, if Shuffler naïvely began fixing code pointers while making code copies with `memcpy`, it would at some point break the `memcpy` implementation, because the latter uses code pointers for a jump table.[2] Hence, we would have to call new functions as they became available, and carefully order the function-pointer rewrite process to avoid invalidating any functions currently on the call stack.

Instead, we opted for a simpler and more general solution. Shuffler is split into two stages, a minimal and a runtime stage. The minimal stage is completely self-contained, and it can safely transform all other code, including `libc` and the second-stage Shuffler. Then it jumps to the shuffled second stage, which erases the previous stage (and all other original code). The second stage inherits all the data structures created in the first so that it can easily create new shuffled code copies. From this point on, Shuffler is fully self-hosting.

4 Implementation

Shuffler runs in userspace on x86-64 Linux. It shuffles binaries, all the shared libraries that a binary depends on, as well as itself. The shuffling process runs asynchronously in a thread, without impeding the execution of the program's threads. Figure 2 shows a running snapshot of shuffled code. Code pointers are directed through the code pointer table and return addresses are stored on the stack, encrypted with an XOR cipher. In each shuffle period, Shuffler makes a new copy of code, updates the code pointer table and sends a signal to all threads (including itself); each thread unwinds and fixes up its stack. Shuffler waits on a barrier until all threads have finished unwinding, then erases the previous code copy.

Our Shuffler implementation supports many system-level features, including shared libraries, multiple threads, forking (each child gets it own Shuffler thread), `{set,long}jmp`, system call re-entry, and signals. Shuffler does not currently support `dlopen` or C++ exceptions. Yet, it does expose several debugging features, notably, exporting shuffled symbol tables to GDB and printing shuffled stack traces on demand.

[2]This crash took place in an earlier prototype of Shuffler.

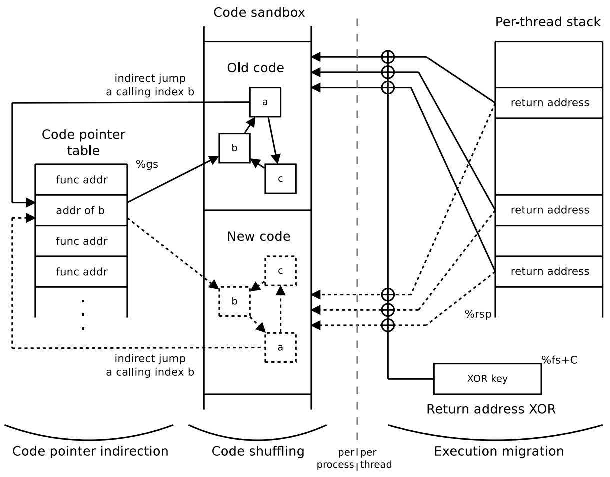


Figure 2: Overview of shuffled code at runtime, as Shuffler executes a shuffle pass. The old code is shown with solid lines and the new code with dotted lines.

Source instruction		Transformation
`lea` *`funcptr`*`, %rax`	→	`lea` *`index`*`, %rax`
`call *%rax`	→	`callq *%gs:(%rax)`

`callq *(%rax,%rbx,8)`	→	`mov (%rax,%rbx,8),%r11`
		`callq *%gs:(%r11)`

`jmp *%rax`	→	`jmpq *%gs:(%rax)`

`jmpq *(%rax,%rbx,8)`	→	`mov %r11, %fs:0x88`
		`mov (%rax,%rbx,8),%r11`
		`mov %gs:(%r11),%r11`
		`xchg %r11, %fs:0x88`
		`jmpq *%fs:0x88`

(a) Transforms to support the code pointer table.

Source instruction		Transformation

`# function begin`	→	`mov %fs:0x28,%r11`
		`xor %r11,(%rsp)`
		`# function begin`

`ret / jmp *%rax`	→	`mov %fs:0x28,%r11`
		`xor %r11,(%rsp)`
		`ret / jmp *%rax`

`call` *`anything`*	→	`call` *`anything`*
		`mov $0x0, -8(%rsp)`

(b) Transforms to support return address encryption.

Figure 3: Binary rewriting transformations performed by Shuffler. `%fs:0x28` is the stack canary, `%r11` is a scratch register, and `%fs:0x88` is a scratch variable.

4.1 Transformations to Support Shuffling

Code pointer abstraction We allocate the code pointer table at load-time and set the base address of the `GS` segment (selected by the `%gs` register) at it. Then, we transform every function pointer at its initialization point from an address value to an index into this table. We use relocations generated by the compiler and preserved by the linker flag `-q` to find all such code pointers. Pointer values are deduplicated as they are assigned indices in the table, for more efficient updating. Jump tables are handled similarly, with indices assigned to each offset within a function that is used as a target. Note that indices may also be assigned dynamically by Shuffler (*e.g.,* so that `setjmp` works across shuffle periods).

We must also transform the code so that indices are invoked properly. As shown in the Figure 3a, every instruction which originally used a function pointer value is rewritten to instead indirect through the `%gs` table. This adds an extra memory dereference. Since x86 instructions can contain at most one memory reference, if there is already a memory dereference, we use the caller-saved register `%r11` as scratch space. For (position-dependent) jump tables, there is no register we can safely overwrite, so we use a thread-local variable allocated by Shuffler as a scratch space (denoted as `%fs:0x88`).

Return-address encryption We encrypt return addresses on the stack with a per-thread XOR key. We reuse the stack canary storage location for our key; our scheme operates similarly to stack canaries, but does not affect the layout of the stack frame. As shown in Figure 3b, we add two instructions at the beginning of every function (to disguise the return address) and before every exit jump (to make it visible again); after each `call`, we insert a `mov` instruction to erase the now-visible return address on the stack. We again use `%r11` as a scratch register, since it is a caller-saved register according to the x86-64 ABI, and thus safe to overwrite.

Displacement reach A normal call instruction has a 32-bit displacement and must be within ± 2GB of its target to "reach" it. Shared libraries use Procedure Linkage Table trampolines to jump anywhere in the 64-bit address space. We wish to use only 32-bit calls and still enable function permutation; thus, we place all shuffled code at most 2GB apart, and transform calls through the PLT into direct function calls. Essentially, we convert dynamically linked programs into statically linked ones at runtime.

4.2 Completeness of Disassembly

We demonstrate the complete and precise disassembly of binaries that have been augmented with a symbol table and relocations. The techniques shown here are sufficient to analyze `libc`, `libm`, `libstdc++`, the SPEC CPU binaries, and the programs listed in our performance evaluation section. While shuffling these libraries and programs, we encountered myriad special cases. Figure 4 lists the main issues we faced, which would also need to be handled by other systems performing similar analyses. The issues boil down to: (a) dealing with inaccurate/missing metadata, especially in the symbol table; (b) handling special types of symbols and relocations; and (c) discovering jump table entries and invocations.

Issue	Description	How to handle
Missing symbol sizes	Internal GCC functions have a symbol size of zero.	Hard-code sizes; `_start` is 42 bytes.
Fall-through symbols	Functions implicitly fall through to the following function.	Attach a copy of the following code.
Overlapping symbols	Some functions are a strict subset of an enclosing function.	Binary search for targets very carefully.
Symbol aliases	Symbol tables have many names for the same function.	Pick one representative name.
Ambiguous names	One LOCAL name, multiple versions (`bsloww` in libm).	Look up address resolved by the loader.
Pointers to static functions	For pointers to functions within the same module, the offset is known, and object files contain no relevant relocations.	Determine if `lea` instructions target a known symbol (not completely sound).
`noreturn` function calls	GCC always generates a NOP after calls to `noreturn` functions like `longjmp`, but omits unwind information.	Detect when at a NOP following a call and use unwind info from at the call.
COPY relocations	Object initialized in one library, then `memcpy`'d to another.	Track data symbols, not just code.
IFUNC symbols	Return pointer to actual function to call (cached in PLT).	Statically evaluate from `lea` refs.
Conditional tail recursion	Does not appear in normal GCC-generated code. Used in hand-coded assembly by glibc (`lowlevellock.h`).	Can do XOR'ing both before and after, works whether or not the jump is taken.
Indirect tail rec.	Difficult to tell apart from jump-table jumps.	Use a function epilogue heuristic.
Finding jump tables	Jump tables are not clearly delineated.	See the text for a discussion on this.

Figure 4: Special cases in augmented binary disassembly.

Jump tables One major challenge is identifying whether relocations are part of jump tables, and distinguishing between indirect tail-recursive jumps and jump-table jumps. If we fail to realize a relocation in a jump table, we will calculate its target incorrectly and the jump will branch to the wrong location; if we decide that a jump table's jump is actually tail recursive, we will insert return-address decryption instructions before it, corrupting `%r11` and scrambling the top of the stack.

GCC generates jump tables differently in position-dependent and position-independent code (PIC). Position-dependent jump tables use 8-byte direct pointers, and are nearly always invoked by an instruction of the form `jmpq *(%rax,%rbx,8)` at any optimization level. PIC jump tables use 4-byte relative offsets added to the address of the beginning of the table—and the `lea` that loads the table address may be quite distant from the final indirect jump. To find PIC jump tables, we use outgoing `%rip`-relative references from functions as bounds and check if they point at sequences of relocations in the data section.[3] Note that `R_X86_64_PC32` relocations must have 4 bytes added to their value (the displacement size) if present in an instruction, and they must not if present in a jump table.

It is difficult to tell whether a `jmpq *%rax` instruction is used for indirect tail recursion, or a PIC jump table. In our system, we must distinguish these to decide whether to decrypt the return address or not. We do this with a heuristic that pairs function epilogues with function prologues. We use a linear sweep to record `push` instructions in the function's first basic block, and keep a log of the `pop` instructions seen since the last jump (within a window size). If an indirect jump is preceded by `pop` instructions that are in the reverse order of the `push` instructions, we assume we have found a function epilogue and that the jump is indirect tail recursive.

4.3 Bootstrapping and Requirements

We carefully bootstrap into shuffled code using two libraries (stage 1 and stage 2) so that the system never overwrites code pointers for the module that is currently executing. These libraries are injected into the target using `LD_PRELOAD`.[4] Rather than reimplement loader functionality, we defer to the system loader to create a valid process image, and then take over before the program—or even its constructors—begin executing.

The constructor of stage 1 is called before any other via the linker mechanism `-z initfirst`.[5] Then, by setting breakpoints in the loader itself, stage 1 makes sure all other constructors run in shuffled code. The last constructor to be called (a side effect of `LD_PRELOAD`) is stage 2's own constructor; stage 2 creates a dedicated Shuffler thread, erases the original copy of all other code, and resumes execution at the shuffled ELF entry point.

4.3.1 Full Shuffling Requirements

Compiler flags We require the program binary and all dependent libraries to be compiled with `-Wl,-q`, a linker flag that preserves relocations. Since we require symbols and DWARF unwind information, the user must avoid `-s`, which strips symbols, and `-fno-asynchronous-unwind-tables`, which elides DWARF unwind information. For simplicity, we do not support some DWARF 3 and 4 opcodes, so the user may need to pass `-gdwarf-2` when compiling

[3]Fortunately, GCC only emits jump tables of size five or more, which makes this heuristic very accurate.

[4]`LD_PRELOAD=./libshuffle0.so:./libshuffle.so`

[5]We require a patch to fully use this mechanism; see Section 4.3.1.

C++. Finally, we found that some SPEC CPU programs required `-fno-omit-frame-pointer`, due to a limitation in our DWARF unwind implementation.

System modifications The `-z initfirst` loader feature currently only supports one shared library, and `libpthread` already uses it. To maintain compatibility with `libpthread`, we patched the loader to support constructor prioritization in multiple libraries. Our 24-line patch transforms a single variable into a linked list. (We have submitted our patch to `glibc` for review.)

Since shuffled functions must be within $\pm$ 2GB of each other, we simplify Shuffler's task and map all ELF `PT_LOAD` sections into the lower 32 bits of the address space (1-line change to the loader). Since `glibc` and `libdl` refer directly to variables in the loader with only 32-bit displacements, we also place the loader itself into that region, preresolving its relocations with `prelink` [3]. Finally, we disabled a manually-constructed jump table in the `vfprintf` of `glibc`, which used computed `goto` statements (1-line change). No other library changes were necessary.

4.4 Implementation Optimizations

Generating new code The Shuffler thread maintains a large code *sandbox* that stores shuffled (and currently executing) functions. In each shuffle period, every function within the sandbox is duplicated and the old copies are erased. The sandbox is split in half so that one half may be easily erased with a single `mprotect` system call.[6] Performance suffers if each function is written to an independent location in the sandbox. The bottleneck is in issuing many `mprotect` system calls (we do not want to expose the whole sandbox by making it writable). Instead, we maintain several *buckets* (64KB–1MB) and each function is placed in a random bucket; when a bucket fills up, it is committed with an `mprotect` call and a fresh bucket is allocated. The Memory Protection Keys (MPK) feature on upcoming Intel CPUs [16] may allow buckets to be created even more efficiently.

Generating function addresses with high entropy (*i.e.,* uniformly at random) is a challenging task. The simplest allocator would pick random addresses repeatedly until a free location is found, but this may require many attempts due to fragmentation. Instead, we use a Fenwick Tree (or Binary Indexed Tree) [30,32] for our allocations. Our tree keeps track of all valid addresses for new buckets, storing disjoint intervals; it also tracks the sum of interval lengths (*i.e.,* the amount of free space). We can select a random number less than this sum and be assured that it maps to some valid free location, and compute this mapping in logarithmic time. This guarantees that each allocation is selected uniformly at random.

Stack unwinding Stack unwinding is performed by parsing the DWARF unwind information from the executable. This information is used by exception handling code, and by the debugger to get accurate stack traces. We found that the popular library `libunwind` [35] was quite unwieldy, used unwind heuristics, and made it difficult to add an address-translation mechanism. Hence, we wrote a custom unwind library with a straightforward DWARF state machine, using binary search to translate between shuffled and original addresses. We generate DWARF information for new code inserted through binary rewriting, and also record the points where return addresses are (or are not) encrypted.

Binary rewriting Shuffler's load-time transformations are all implemented through binary rewriting. We disassemble each function with diStorm [21] and produce intermediate data structures which we call *rewrite blocks*. Rewrite blocks are similar to basic blocks but may be split at arbitrary points to accommodate newly inserted instructions. Through careful block splitting, we can choose whether incoming jumps execute or skip over new instructions as appropriate. This data structure also allows fast linear updates of internal offsets for jump instructions. We promote 8-bit jumps to 32-bit jumps (iteratively) if the jump targets have become too far away. Once jumps and other data structures are consistent, the final code size is known and we create the first shuffled copy of a function. The runtime shuffling process copies the shuffled version of each function to a new location and patches it without invoking the rewriting procedure.

5 Performance Evaluation

Unless otherwise noted, performance results were measured on a dual-socket 2.8GHz Westmere Xeon X5660 machine, with 64GB of RAM and 24 cores (hyperthreading enabled), running Ubuntu 16.04 with GCC 4.8.4.

5.1 SPEC CPU2006 Overhead

We ran Shuffler on all C and C++ benchmarks in SPEC CPU2006, over a range of different shuffling periods. The SPEC baseline was compiled with its default settings (`-O2`). The shuffled versions were compiled the same way with the addition of `-Wl,-q` (see Section 4.3.1), and also `-fno-omit-frame-pointer` due to a limitation in our DWARF unwind implementation. Since Shuffler does not yet support C++ exceptions, we replaced exceptions with conventional control flow in `omnetpp` (20-line change) and `povray` (15 lines).

Effect of shuffling rate Figure 5 shows the overhead observed by the single-threaded SPEC benchmarks at different shuffling rates, excluding the overhead of the

[6]This also clears the old code from the instruction cache, since Linux's updates to the Translation Lookaside Buffer (TLB) flush the appropriate cache lines as per Section 4.10.4 of the Intel manual [39].

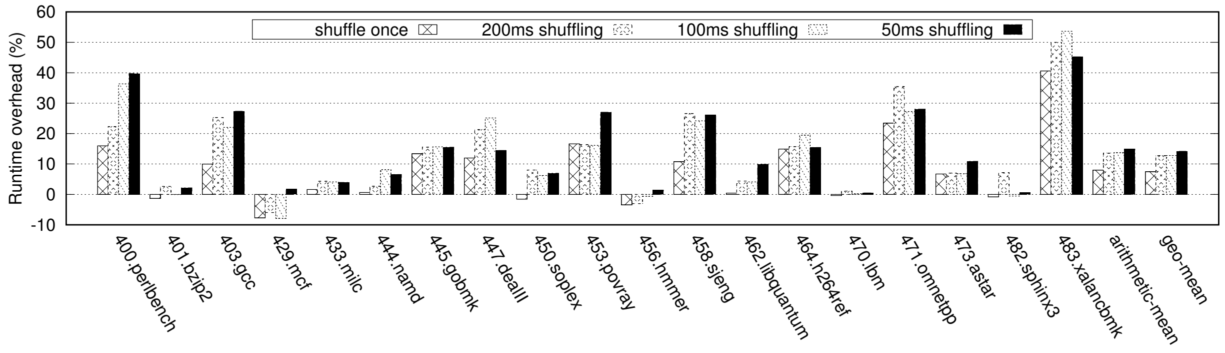


Figure 5: Shuffler performance (shown as overhead percentage) on SPEC CPU2006 at different shuffling rates.

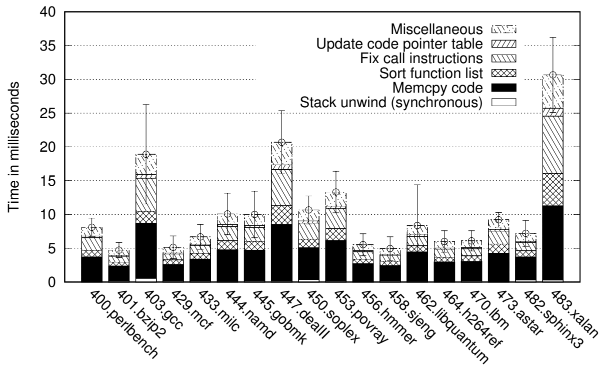


Figure 6: SPEC CPU continuous shuffling breakdown. Synchronous (stack unwind) overhead is barely visible at the bottom. Data for `omnetpp` was not gathered.

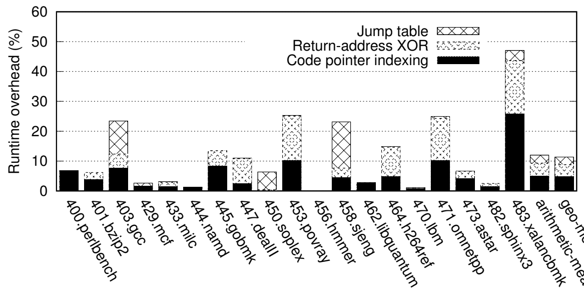


Figure 7: Static transformation overheads in SPEC CPU.

Shuffler thread. The average overheads are 7.99% (shuffling once), 13.5% (200ms shuffling), 13.7% (100ms shuffling), and 14.9% (50ms shuffling). Considering that thousands of shuffles were performed in each case (the runtime per program is from 3.5–10 minutes), the observed overhead is acceptable. Note that faster shuffling rates do not cause significant slowdown, because the static code rewriting cost is paid only once (up-front).

Asynchronous overhead By design, Shuffler offloads the majority of the shuffling computations onto another CPU core (see Figure 6). We assume that the protected system is not at full capacity and has sufficient cycles to execute the Shuffler thread concurrently.

We can, however, approximate the shuffling overhead: the asynchronous shuffling time divided by the shuffling period yields the CPU load. Assuming `gcc` asynchronously shuffles in 25 milliseconds, it would use 50% of the offload core in a shuffle period of 50 milliseconds, and 25% in a shuffle period of 100 milliseconds. We confirmed this approximation by measuring the reported CPU usage once per second, as each SPEC CPU program ran. The true overheads were within a few percentage points of the approximation. For instance, `xalancbmk` was predicted to use 61.31% of the CPU in the Shuffler thread and in fact used 58.64%. This overhead is examined in more detail in Section 5.2.

Synchronous overhead The only synchronous work in Figure 6 is the short time when the program thread is interrupted via a signal to perform stack unwinding. Shuffler's stack unwind performance is linear in the call stack depth, processing 3247 stack frames per millisecond (including the thread barrier synchronization time between Shuffler and the program threads). Most SPEC programs have modest call stack depths, except `xalancbmk`, where certain stages have call stacks at least 20,000 deep (up to 45,000), and take up to 6 ms to unwind. The highest average unwind time is 0.53 ms for `gcc`; the Shuffler thread unwinds itself in ∼0.025 ms.

5.1.1 Static overhead on SPEC CPU

In Figure 7, we break down the overhead observed due to static code transformations (when only shuffling once). This overhead is purely from the inserted instructions. The average overhead is 2.68% due to jump table rewriting, 4.36% due to return address encryption, and 4.78% due to code pointer abstraction. Jump table numbers are relative to a baseline with jump tables; everything else, to one without (the baselines only differ by 0.45%).

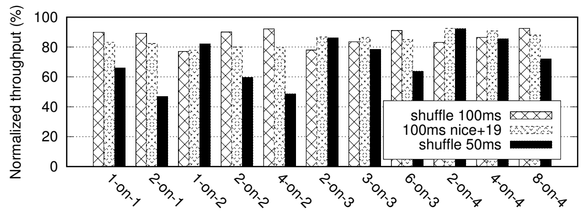
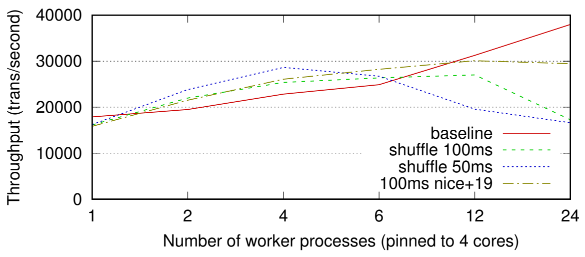
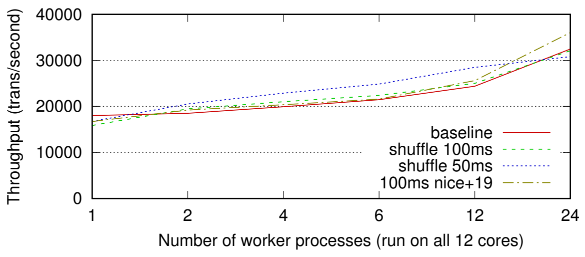


Figure 8: Shuffler thread impact on Nginx throughput. *t*-on-*n* means *t* worker processes pinned to *n* cores.



(a) Nginx workers and Shuffler threads pinned to 4 cores.



(b) Shuffled Nginx running on all 12 available cores.

Figure 9: Shuffled Nginx performance at a larger scale.

Jump tables Jump table overhead can be high, because our transformation to support code pointer indices is inefficient for position-dependent jump tables (see Section 4.2). With greater compiler integration or more thorough binary rewriting, this overhead can be reduced.

Return-address encryption The return-address encryption overhead increases as the program makes more function calls. The 4.36% overhead is higher than for a straightforward stack canary scheme. However, it also provides disclosure resilience for return addresses, which is essential for our method. Other strong shadow stack schemes are available [22], with comparable performance. We could use dynamically allocated table indices for return addresses, but disrupting `call`/`ret` pairs has high performance overhead [22, 43].

Code pointer abstraction The code pointer abstraction overhead is high when the program makes a large number of indirect calls. For instance, `xalancbmk` makes 3.35 million indirect calls on the test input size, 3.60 billion calls on train, and likely an order of magnitude more on ref. This overhead is mostly unavoidable; the layer of indirection introduced by these transformations is what allows Shuffler to invalidate old code addresses without using (code) pointer tracking. We confirmed with the Linux `perf` tool that the percentage overhead from code pointer abstraction corresponds to the percentage of the newly inserted instructions.

5.2 Nginx Overhead

We ran performance experiments on the Nginx 1.4.6 web server. Our setup used two dual hex-core machines on a dedicated gigabit network, each with Turbo mode and hyperthreading disabled (hence 12 cores each). The client machine was the same one used for SPEC CPU, and the server had two 2.50GHz Xeon E5-2640 CPUs.

To generate client load, we used the multithreaded Siege [31] benchmarking tool. We used a request size of 100 bytes with 32 concurrent connections. This configuration ensures that the server is CPU-bound; larger sizes may exceed network bandwidth, while more connections cause CPU scheduling delays on the client machine. Measurements are reported as the average of five 30-second runs. Siege reported a latency of less than 10 milliseconds, and a concurrency level between 30.86 and 31.76, for all baseline and shuffled test cases.

Shuffler thread overhead First, we investigated the performance of Shuffler threads in Nginx. In the beginning, Nginx has one master process and one Shuffler thread, and then it forks into a user-specified number of worker processes (each with their own Shuffler thread). In our evaluation, we pinned all Nginx workers and their associated Shuffler threads to a case-dependent number of cores, and excluded the master and its Shuffler thread by pinning them to a different core on the same socket.

The results are shown in Figure 8. In the 1-on-1 case, there is one Nginx worker process and its Shuffler thread on a single core. These two threads will compete for scheduling time slices on the same core, and whenever the Shuffler thread is scheduled, throughput is stalled (since Nginx can only run on the same core). Shuffler takes about 15 milliseconds to shuffle Nginx, so we would expect 15% slowdown at 100 millisecond shuffling and 30% slowdown at 50 millisecond shuffling. The measurements track this expectation quite closely.

Some cases have greater overcommitting, *e.g.,* 4-on-2 has four Nginx workers plus four Shuffler threads on two cores. Overhead is still reasonable, and the throughput is around 85%-90% of the baseline. Setting the Shuffler threads to lower priority (nice +19) at 100 ms does not increase throughput here, although it does help when a greater portion of the system is in use (see below).

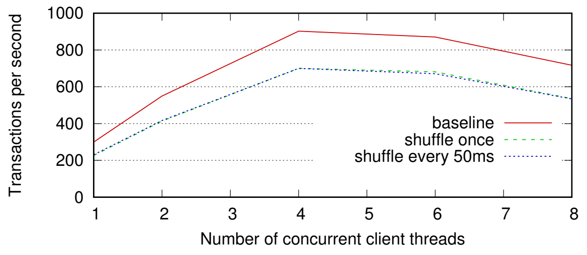


Figure 10: MySQL transaction throughput as measured by SysBench. Shuffle once and shuffle every 50ms incurs the same overhead.

Full-scale Nginx overhead In our second set of Nginx experiments, we pinned all threads (including the master process) to a certain number of cores. Figure 9a shows the results when pinned to four cores on the same socket, and Figure 9b shows the results with no pinning (*i.e.,* all 12 cores available for scheduling). In the four-core case, the overhead starts to get very high with 12 and 24 workers. This is because the Linux scheduler must try to place all worker threads, Shuffler threads, and the master (for a total of 26 or 50 threads) onto a mere four cores. To assist the scheduler, we made each Shuffler thread set its nice value to +19 (low priority) at 100 ms, which results in longer shuffling latencies but greater throughput since Nginx worker threads get more CPU time.

In the case of no CPU pinning (Figure 9b), Shuffler performance tracks the baseline very well. There is less overcommitting here: even in the 24 worker case, each core has two workers and two Shuffler threads to schedule. In the nice+19 case, shuffling latencies (for 24-on-12) are high with average 18.1 ms and std. dev. 266, instead of the original average 17.4 ms, std. dev. 39. Overall, we measured small speedups over the baseline, which is likely experimental noise; Shuffler threads do not significantly impact the overall system performance. This full-system experiment incorporates the master process overhead, as well as kernel I/O threads, which normally ignore userspace CPU pinning (and use idle cores).

5.3 Other Macro Benchmarks

MySQL We shuffled MySQL continuously every 50 ms (asynchronous shuffling takes 30 ms), querying its 10 million row database using SysBench on localhost. The machine had 24 cores and MySQL used the default of 16 threads. Figure 10 shows that the performance overhead (30.9%) is almost completely due to static rewriting, and shuffling every 50ms has the same performance as shuffling once. This is partially because unlike Nginx, where workers are separate processes and thus require separate Shuffler threads, MySQL worker threads are all randomized by a single Shuffler thread.

Program	Code + Syms/Relocs	Data Structs + Overhead
Shuffler	0.16MB + 0.15MB	(included below)
SQLite	2.20MB + 1.63MB	32.2MB + 23.7MB
Nginx	3.14MB + 2.68MB	45.7MB + 37.7MB
Xalan	4.36MB + 5.09MB	76.7MB + 44.3MB

Figure 11: Program size and Shuffler overhead.

So using multithreaded workers instead of multiprocess workers can amortise Shuffler's performance overhead, with an appropriate tradeoff in security (see Section 6.2).

SQLite SQLite has a reasonably small codebase which only takes the Shuffler thread 5 milliseconds to shuffle. We shuffled it at 20 ms for a week without incident.

Mozilla's SpiderMonkey We shuffled the JavaScript engine SpiderMonkey and it passed its test suite of 3600 test cases. We had to disable JIT code generation (IonMonkey); Shuffler could in future handle JIT code if it was informed of when new code chunks were generated.

5.4 Memory Overhead

Figure 11 reports the code/relocation/symbol section sizes for programs and their libraries. Shuffler's total memory overhead consists of: an in-flight copy of all code sections; the code pointer table (1MB); one signal stack (64KB) per thread; metadata structures like relocation and symbol hash tables; and the current permuted list of functions (32 bytes per function). For allocation efficiency, code copies are stored in a preallocated 160MB sandbox. We use a custom `malloc` implementation [41], and report its bookkeeping/fragmentation overhead separately. The permuted function list is destroyed and recreated for each shuffle period.

5.5 TASR Performance Comparison

The closest re-randomization system to Shuffler is TASR [7], which has a reported overhead of 0–10% (2.1% average) on SPEC CPU. However, those numbers are against a baseline compiled with `-Og`, which only performs optimizations that preserve debugging information. Such optimizations are fairly limited: we found that SPEC CPU with `-Og` is 30% slower than with the normal optimization level `-O2`. In other words, TASR's performance overhead is 30-40% relative to the true baseline (while Shuffler's is under 15%). Unfortunately, using `-Og` is intrinsic to any scheme like TASR that requires accurate tracking of source-level variables.

Additionally, TASR's scheme of randomizing on I/O system call pairs provides strong guarantees, but seems unlikely to scale to real-world server applications. In the case of Nginx, we measured that processing a 100KB request takes 0.22 milliseconds. Let us assume that TASR can randomize Nginx in 15 milliseconds (note that this is Shuffler's rate—TASR is likely to take even longer since it injects and runs a pointer updater process). Since TASR re-randomizes after each request, it would incur 15 milliseconds of latency per 0.22 milliseconds of useful work, resulting in 1.5% of the original throughput. The scheme could be extended to allow multiple requests to run in parallel, but this would still require 68 threads on 68 cores to maintain the original throughput.

6 Security Analysis

In this section we show how Shuffler defends against existing attacks assuming all its mechanisms are in place, including code pointer indirection, return address encryption, and continuous shuffling every r milliseconds. Then we discuss other possible attacks against the Shuffler infrastructure, and follow up with some case studies.

6.1 Analysis of Traditional Attacks

Normal ROP It is fairly obvious that a traditional ROP attack will fail when the target is being shuffled, because the addresses of gadgets are hard-coded into the exploit. Shuffler's code sandbox currently has 27 bits of entropy (a 31-bit sandbox should be possible as per Section 4.1) and gadgets could be anywhere in the sandbox. Thus, if the ROP attack uses N distinct gadgets, the chance of it succeeding is approximately 2^{-27N}. Any attack which desires better odds needs to incorporate a memory disclosure component to discover what Shuffler is doing.

Indirect JIT-ROP Indirect JIT-ROP relies on leaked code pointers and computes gadgets accordingly. Because code pointers are replaced with table indices, the attacker cannot gather code pointers from data structures; nor can the attacker infer code pointers from data pointers, since the relative offset between code and data sections changes continuously. While the attacker can disclose indices, these are not nearly as useful as addresses: they can only be used to jump to the beginning of a function, and they cannot reveal the locality of nearby functions. We assume indices are randomly ordered at load time, with gaps (traps) in the index space to prevent an attacker from easily brute-forcing it [18]. The table itself is a potential source of information, but the table's location is randomized and it is continuously moved (see Section 6.2 below). Return addresses are encrypted with an XOR cipher, so disclosing them does not reveal true code addresses. In fact there are no sources of code pointers accessible to an attacker by way of memory disclosure, and so indirect JIT-ROP is impossible by construction.

Direct JIT-ROP In direct JIT-ROP [55], the attacker is assumed to know one valid code address, and employs a memory disclosure recursively, harvesting code pages and finding enough gadgets for a ROP attack. A control flow hijack is used to kick off the exploit execution.

Our argument against JIT-ROP is threefold. First, the attacker must be able to obtain the first valid code address, and as described for indirect JIT-ROP, there is no accessible source of code pointers in the program. Thus the attacker must resort to brute force or side channels (as for Blind ROP below). Second, once an attack has been completely constructed, there is no easy way to jump to an address of the attacker's choosing: indirect calls and jumps treat their operands as table indices, not addresses, while return statements mangle the return address before branching to a target. The attacker must therefore use a partial return address overwrite (described below in Section 6.2), which itself has a significant chance of failure.

Thirdly, and most importantly, the entire attack must be completed within the shuffle period of r milliseconds. No useful information carries over from one shuffle period to the next, and all previously discovered code pages and gadgets are immediately erased. If the attacker can do everything in r milliseconds, they win; thus, the defender should select a small enough r to disrupt any anticipated attacks. We discuss the attack time required in Section 6.3. The fastest published attack times are on the order of several seconds, not tens of milliseconds.

Blind ROP Blind ROP [8] tries to infer the layout of a server process by probing its workers, which are `fork`ed from the parent and have the same layout. The attack uses a timing channel to infer information about the parent based on whether the child crashed or not. Shuffler easily thwarts this attack because it randomizes child and parent processes independently.

6.2 Shuffler-specific Attacks

Breaking XOR encryption Our XOR encryption is less vulnerable to brute force than typical XOR ciphers. Leaking multiple return addresses does not allow the attack to easily construct linear relations, because there are two unknowns: random values (addresses) encrypted under a random key. The addresses are re-randomized during each shuffle period, and the XOR key could be too. If every function uses it own key, the attacker's task becomes even harder [10]. The keys are stored at unknown addresses in thread-local storage. While there is a small window of two instructions after calls during which the unencrypted return address is visible on the stack, this would be difficult to exploit because the attacker cannot insert any intervening instructions—though a determined attacker might try to do so from another thread.

It is possible to bypass XOR in other ways. For example, an attacker might partially overwrite an encrypted return address, attempting to increment the return address by a small amount without knowing the plaintext value. This could be used to initiate execution of a misaligned gadget, or to trampoline through a return instruction and jump straight to an attacker-controlled address. Such an attack would be difficult; the attacker would need to find a function on the call stack with appropriate known code layout, and then brute-force several bits of the canary.

Ciphertext-only attacks The attacker could attempt to swap valid code pointer indices. This allows an attacker to jump to the beginning of functions whose address is taken, similar to the restrictions under coarse-grained Control Flow Integrity (CFI) [61, 62]—and such defenses have been bypassed [23, 36]. The mapping between indices and functions would have to first be discovered (subject to permutation and traps). We consider this a data-only attack [12]. As per Section 2.1, we do not attempt to add to the literature for data-only attacks.[7]

The attacker might swap valid encrypted return addresses on the stack. This is equivalent to jumping to call-preceded gadgets (as in coarse-grained CFI), but using only those functions which occur on the call stack. While such an attack may be theoretically possible, it has not been demonstrated in the literature—especially within the constraints of a single shuffle period, where return addresses change every *r* milliseconds.

Parallel attacks When Shuffler is defending a multi-threaded program, every thread uses the same shuffled code layout. Thus, an attacker might run a parallel disclosure attack, multiplying the information that may be gathered from a single-threaded program. However, parallel disclosure is limited by dependencies—often one page's address is computed from another's content, so the disclosures are not parallelizable. In the worst case, defending a parallel attack requires a linearly faster shuffling rate. Currently, the user can run a multiprocess program instead (like Nginx) to avoid this issue. We also used the `%gs` register to store our code pointer table intentionally so that code could be shared between threads. It would be fairly straightforward to use the thread-local `%fs` register instead to maintain separate code copies and pointer tables for each thread, at a corresponding increase in memory and CPU use.

Exploiting the Shuffler infrastructure Since Shuffler runs in an egalitarian manner in the same address space as the target, it may be vulnerable to attack. Shuffler's code is shuffled and defended in the same way as the target, and any specific functionality (*e.g.,* dynamic index allocation) is not accessible through static references. However, Shuffler's data structures might be disclosed at runtime—*e.g.,* to reveal the location of every chunk of code. We are careful to place sensitive information in exactly one data structure, the list of chunks, which is itself destroyed and moved in each shuffle period. There is a single global pointer to this list, which is stored in the `%gs` table along with code pointers.

[7] Thwarting this means updating indices at runtime; see Section 3.2.

Shuffler's code pointer table might itself be used to execute functions, or read or write function locations. As described earlier in Section 6.1, we assume that the table contains traps or invalid entries. This impedes execution of gadgets and requires the index-to-code mapping to be unravelled first. However, the table can be read and written directly with `%gs`-relative gadgets—which are not used by shuffled code but may occur at misaligned offsets. Writes can be disallowed using page permissions. Reads yield information that is only useful for one shuffle period; it is also a "chicken-and-egg" problem to rely on such a gadget to find one's gadgets.

Although the table contains many addresses that the attacker would like to disclose, we assume that the table location is randomized and is continuously moving during the shuffling process. The table's location is only stored in kernel data structures and the inaccessible model-specific register `%gs`. While x86 has a new instruction to read `%gs`, called `RDGSBASE`, it must be enabled through processor control flags (Linux v4.6 does not support that feature). Thus, the attacker must find the table's location through cache timing attacks or allocation spraying [37, 48], which has not been shown to be effective against a continuously moving target.

Finally, even if all of Shuffler's data is disclosed, the addresses for the next shuffle period can be made unpredictable by reseeding Shuffler's random number generator with the kernel-space PRNG `/dev/urandom`.

Shuffler thread compromise If the Shuffler thread crashes for whatever reason, the target program could continue executing its current copy of code unhindered (and undefended). To guard against this, we install signal handlers for common fatal signals. Our default policy is to terminate the process if a crash occurs in Shuffler code. We could also attempt to restart the Shuffler thread (as is done on fork). Instead of causing an outright crash, the attacker could attempt to hang the Shuffler thread, *e.g.,* by pretending that another thread has been created through data structure corruption. This particular technique would cause all threads to hang in the post-unwind synchronization barrier, inside Shuffler code, which is not very useful for an attacker. Still, if a user is concerned that the Shuffler thread may be compromised, an external watchdog can periodically ensure (*e.g.,* by examining `/proc/<pid>/maps`) that shuffling is still occurring.

6.3 Case Studies

Disclosing memory pages When conducting a JIT-ROP attack, the attacker has a tradeoff: either quickly scan memory pages for desired gadgets, which may require many source pages; or, spend more time looking for gadgets in a small number of pages, which can be computationally prohibitive. The original JIT-ROP [55] attack searches through 50 pages to find the gadgets for an attack, and takes 2.3–22 seconds to carry out a full exploit. The ROP compiler Q [52] can attack executables as small as 20KB, but due to their use of heavyweight symbolic execution and constraint solving, their published real-world attack computation times are 40–378 seconds.

Fetching pages takes time because real memory disclosures do not execute instantaneously. The original JIT-ROP [55] attacks can harvest 3.2, 22.4, and 84 pages/second (*e.g.,* requiring between 12 and 312 milliseconds per page). We reproduced Heartbleed on OpenSSL 1.0.1f using Metasploit [45] and found that the attack takes 60ms to complete (17.2ms per additional disclosure), when the attacker is on the local machine.

Network communication latency For server programs, the network communication latency must be added to every memory disclosure's execution time. According to data from WonderProxy [50], long-distance packet speeds are about 22% the speed of light. We tested this by communicating between servers on the east and west coast of the United States, observing 65.94 and 67.57 ms ping times where 59.27 was predicted. Thus, every millisecond of round-trip ping implies a physical separation of 41 miles (66 km). For example, to perform a single disclosure and then a control-flow hijack against a server shuffled every 20 milliseconds, the attacker would need to be within 820 miles (1320 km).

Continuous re-randomization ensures that addresses are only valid for a short time period. One could eliminate this time window entirely by introducing artificial latency for requests. Each request response would be held in an outgoing queue until a re-randomization has occurred—increasing the server's latency, but guaranteeing that all leaked information is already out-of-date.

Small-scale JIT-ROP attack We created a small vulnerable server to simulate a JIT-ROP scenario. The program prints its stack canary and a known code address, using inline assembly to read the code pointer table. We have an 8-byte memory disclosure (a request which overruns a buffer and corrupts a pointer). We use this vulnerability repeatedly to leak a full 4KB page (which takes 8 milliseconds over loopback). Finally, we overwrite a return address to point at a leaked function. With 8 millisecond shuffling or faster, the attack crashes the target; at slower shuffling rates, the attack succeeds.

Real-world Blind-ROP attack We reproduced the Blind-ROP [8] attack against Nginx 1.4.0 (using CVE-2013-2028 [44]). We measured that the attack takes seven minutes to complete. When Nginx was shuffled, the attack was unable to find the Procedure Linkage Table or stack canary; it received false feedback since parent and child processes are randomized independently.

7 Discussion and Future Work

The commonly accepted wisdom is that performing analysis on binaries is challenging. In fact, while handcrafted binaries can be pathological, compiler-generated code is relatively straightforward to disassemble. Thus, building binary-level defenses is quite possible, especially for symbol- and relocation-augmented binaries.

We are able to perform continuous re-randomization quite efficiently. This is partially because program code size is small, and because the cost of code rewriting is paid only once up-front (not during each shuffle). However, while shuffling in a separate thread is excellent for efficiency, it can lead to unpredictable shuffling latencies, especially under load. Ideally, the target code would need to check in periodically with Shuffler and not run indefinitely. Also, while we currently use a single Shuffler thread, the shuffling process is parallelizable to multiple worker threads if higher shuffling rates are desired.

Most defensive techniques exist outside the infrastructure they defend, or declare themselves part of the trusted computing base. We hope that Shuffler's design will inspire more egalitarian techniques, and in general more techniques that pay attention to their own attack surface.

8 Conclusion

We present Shuffler, a system which defends against all forms of code reuse through continuous code re-randomization. Shuffler randomizes the target, all of the target's libraries, and even the Shuffler code itself—all within a real-time shuffling deadline. Our focus on egalitarian defense allows Shuffler to operate at the same level of privilege as the target, from within the same address space, enabling deployment in environments such as the cloud. We require no modifications to the compiler or kernel, nor access to source code, leveraging only existing compiler flags to preserve symbols and relocations. For the best possible performance, we perform shuffling asynchronously, making use of spare CPU cycles on idle cores. Programs spend 99.7% of their time running unhindered, and only 0.3% of their time running stack unwinding to migrate between copies of code. Shuffler can randomize SPEC CPU every 50 milliseconds with 14.9% overhead. We shuffled real-world applications including MySQL, SQLite, Mozilla's SpiderMonkey, and Nginx. Finally, Shuffler scales well on Nginx, up to a full system load of 24 worker processes on 12 cores.

9 Acknowledgements

We thank the anonymous reviewers, our shepherd Andrew Baumann, and Mihir Nanavati for their valuable comments. This paper was supported in part by ONR N00014-12-1-0166 and N00014-16-1-2263; NSF CCF-1162021, CNS-1054906, and CNS-1564055; an NSF CAREER award; and an NSERC PGS-D award.

References

[1] ABADI, M., BUDIU, M., ERLINGSSON, U., AND LIGATTI, J. Control-flow integrity. In *Proc. of ACM CCS* (2005).

[2] ALEPHONE. Smashing the stack for fun and profit. `https://users.ece.cmu.edu/~adrian/630-f04/readings/AlephOne97.txt`, 1997.

[3] ARCH WIKI. Prelink. `https://wiki.archlinux.org/index.php/Prelink`, 2015.

[4] BACKES, M., HOLZ, T., KOLLENDA, B., KOPPE, P., NÜRNBERGER, S., AND PEWNY, J. You can run but you can't read: Preventing disclosure exploits in executable code. In *Proc. of ACM CCS* (2014).

[5] BACKES, M., AND NÜRNBERGER, S. Oxymoron: Making fine-grained memory randomization practical by allowing code sharing. In *Proc. of USENIX Security* (2014), pp. 433–447.

[6] BHATKAR, S., SEKAR, R., AND DUVARNEY, D. C. Efficient techniques for comprehensive protection from memory error exploits. In *Proc. of USENIX Security* (2005), pp. 271–286.

[7] BIGELOW, D., HOBSON, T., RUDD, R., STREILEIN, W., AND OKHRAVI, H. Timely rerandomization for mitigating memory disclosures. In *Proc. of ACM CCS* (2015), pp. 268–279.

[8] BITTAU, A., BELAY, A., MASHTIZADEH, A., MAZIERES, D., AND BONEH, D. Hacking blind. In *Proc. of IEEE S&P* (2014), pp. 227–242.

[9] BLETSCH, T., JIANG, X., FREEH, V. W., AND LIANG, Z. Jump-oriented programming: a new class of code-reuse attack. In *Proc. of ACM CCS* (2011), pp. 30–40.

[10] BRADEN, K., CRANE, S., DAVI, L., FRANZ, M., LARSEN, P., LIEBCHEN, C., AND SADEGHI, A.-R. Leakage-resilient layout randomization for mobile devices. In *Proc. of NDSS* (2016).

[11] CARLINI, N., BARRESI, A., PAYER, M., WAGNER, D., AND GROSS, T. R. Control-flow bending: On the effectiveness of control-flow integrity. In *Proc. of USENIX Security* (2015), pp. 161–176.

[12] CHEN, S., XU, J., SEZER, E. C., GAURIAR, P., AND IYER, R. K. Non-control-data attacks are realistic threats. In *Proc. of USENIX Security* (2005).

[13] CHEN, Y., WANG, Z., WHALLEY, D., AND LU, L. Remix: On-demand live randomization. In *Proc. of ACM CODASPY* (2016), pp. 50–61.

[14] CONTI, M., CRANE, S., DAVI, L., FRANZ, M., LARSEN, P., NEGRO, M., LIEBCHEN, C., QUNAIBIT, M., AND SADEGHI, A.-R. Losing control: On the effectiveness of control-flow integrity under stack attacks. In *Proc. of ACM CCS* (2015), pp. 952–963.

[15] CORBET, J. x86 NX support. `http://lwn.net/Articles/87814/`, 2004.

[16] CORBET, J. Memory protection keys [lwn.net]. `https://lwn.net/Articles/643797/`, 2015.

[17] CRANE, S., LIEBCHEN, C., HOMESCU, A., DAVI, L., LARSEN, P., SADEGHI, A.-R., BRUNTHALER, S., AND FRANZ, M. Readactor: Practical code randomization resilient to memory disclosure. In *Proc. of IEEE S&P* (2015), pp. 763–780.

[18] CRANE, S. J., VOLCKAERT, S., SCHUSTER, F., LIEBCHEN, C., LARSEN, P., DAVI, L., SADEGHI, A.-R., HOLZ, T., DE SUTTER, B., AND FRANZ, M. It's a TRaP: Table randomization and protection against function-reuse attacks. In *Proc. of ACM CCS* (2015), pp. 243–255.

[19] CURTSINGER, C., AND BERGER, E. D. Stabilizer: Statistically sound performance evaluation. In *Proc. of ACM SIGARCH* (Mar. 2013), pp. 219–228.

[20] CVEDETAILS. Vulnerability distribution of CVE security vulnerabilities by types. `https://www.cvedetails.com/vulnerabilities-by-types.php`, 2016.

[21] DABAH, G. distorm3. `http://ragestorm.net/distorm/`, 2003–2012.

[22] DANG, T. H., MANIATIS, P., AND WAGNER, D. The performance cost of shadow stacks and stack canaries. In *Proc. of ACM CCS* (2015), pp. 555–566.

[23] DAVI, L., LEHMANN, D., SADEGHI, A.-R., AND MONROSE, F. Stitching the gadgets: On the ineffectiveness of coarse-grained control-flow integrity protection. In *Proc. of USENIX Security* (Aug. 2014).

[24] DAVI, L., LIEBCHEN, C., SADEGHI, A.-R., SNOW, K. Z., AND MONROSE, F. Isomeron: Code randomization resilient to (just-in-time) return-oriented programming. In *Proc. of NDSS* (2015).

[25] DEBIAN. Hardening - Debian Wiki. `https://wiki.debian.org/Hardening`, 2015.

[26] DEBIAN. sbuild - Debian Wiki. `https://wiki.debian.org/sbuild`, 2016.

[27] EAGLE, C. *The IDA pro book: the unofficial guide to the world's most popular disassembler*. No Starch Press, 2011.

[28] EVANS, I., LONG, F., OTGONBAATAR, U., SHROBE, H., RINARD, M., OKHRAVI, H., AND SIDIROGLOU-DOUSKOS, S. Control jujutsu: On the weaknesses of fine-grained control flow integrity. In *Proc. of ACM CCS* (2015), pp. 901–913.

[29] FEDORA. Harden All Packages - Fedora Project. `https://fedoraproject.org/wiki/Changes/Harden_All_Packages`, 2016.

[30] FENWICK, P. M. A new data structure for cumulative frequency tables. *Software: Practice and Experience 24*, 3 (1994), 327–336.

[31] FULMER, J. Siege home. `https://www.joedog.org/siege-home/`, 2012.

[32] GEEKSFORGEEKS. Binary indexed tree or Fenwick tree. `http://www.geeksforgeeks.org/binary-indexed-tree-or-fenwick-tree-2/`, 2015.

[33] GIONTA, J., ENCK, W., AND NING, P. HideM: Protecting the contents of userspace memory in the face of disclosure vulnerabilities. In *Proc. of ACM CODASPY* (2015), pp. 325–336.

[34] GIUFFRIDA, C., KUIJSTEN, A., AND TANENBAUM, A. S. Enhanced operating system security through efficient and fine-grained address space randomization. In *Proc. of USENIX Security* (2012), pp. 475–490.

[35] GNU. The libunwind project. `http://savannah.nongnu.org/projects/libunwind/`, 2014.

[36] GÖKTAS, E., ATHANASOPOULOS, E., BOS, H., AND PORTOKALIDIS, G. Out of control: Overcoming control-flow integrity. In *Proc. of IEEE SOSP* (2014).

[37] GÖKTAS, E., GAWLIK, R., KOLLENDA, B., ATHANASOPOULOS, E., PORTOKALIDIS, G., GIUFFRIDA, C., AND BOS, H. Undermining information hiding (and what to do about it). In *Proc. of USENIX Security* (2016).

[38] HISER, J., NGUYEN-TUONG, A., CO, M., HALL, M., AND DAVIDSON, J. W. ILR: Where'd My Gadgets Go? In *Proc. of IEEE SOSP* (2012), pp. 571–585.

[39] INTEL. Intel 64 and IA-32 Architectures Software Developer's Manual Volume 3A: System Programming Guide, Part 1, Mar 2010.

[40] KUZNETSOV, V., SZEKERES, L., PAYER, M., CANDEA, G., SEKAR, R., AND SONG, D. Code-pointer integrity. In *Proc. of USENIX OSDI* (2014), pp. 147–163.

[41] LEE, D. A memory allocator. http://g.oswego.edu/dl/html/malloc.html, 2000.

[42] LU, K., NÜRNBERGER, S., BACKES, M., AND LEE, W. How to make ASLR win the clone wars: Runtime re-randomization. In *Proc. of NDSS* (2016).

[43] MCCAMANT, S., AND MORRISETT, G. Evaluating SFI for a CISC Architecture. In *Proc. of USENIX Security* (2006).

[44] MITRE CORPORATION. CVE-2013-2028. http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2013-2028, 2013.

[45] MOORE, H., ET AL. The Metasploit Project. http://www.metasploit.com/, 2009.

[46] MSDN. Symbols and symbol files - Windows 10 hardware dev. https://msdn.microsoft.com/en-us/library/ff558825.aspx, 2016.

[47] NIU, B., AND TAN, G. Modular control-flow integrity. In *Proc. of ACM PLDI* (2014).

[48] OIKONOMOPOULOS, A., ATHANASOPOULOS, E., BOS, H., AND GIUFFRIDA, C. Poking holes in information hiding. In *Proc. of USENIX Security* (2016).

[49] PAX TEAM. PaX address space layout randomization (ASLR). http://pax.grsecurity.net/docs/aslr.txt, 2003.

[50] REINHEIMER, P. Miles per millisecond: A look at the WonderProxy network. https://wonderproxy.com/blog/miles-per-milisecond/, 2011.

[51] ROGLIA, G. F., MARTIGNONI, L., PALEARI, R., AND BRUSCHI, D. Surgically returning to randomized lib(c). In *Proc. of USENIX ACSAC* (2009), pp. 60–69.

[52] SCHWARTZ, E. J., AVGERINOS, T., AND BRUMLEY, D. Q: Exploit hardening made easy. In *Proc. of USENIX Security* (2011), pp. 25–25.

[53] SCHWARZ, B., DEBRAY, S., AND ANDREWS, G. Disassembly of executable code revisited. In *Proc. of IEEE WCRE* (2002), pp. 45–54.

[54] SHACHAM, H. The geometry of innocent flesh on the bone: Return-into-libc without function calls (on the x86). In *Proc. of ACM CCS* (2007), pp. 552–61.

[55] SNOW, K. Z., MONROSE, F., DAVI, L., DMITRIENKO, A., LIEBCHEN, C., AND SADEGHI, A.-R. Just-in-time code reuse: On the effectiveness of fine-grained address space layout randomization. In *Proc. of IEEE SOSP* (2013).

[56] SOLAR DESIGNER. lpr libc return exploit. http://insecure.org/sploits/linux.libc.return.lpr.sploit.html, 1997.

[57] TANG, A., SETHUMADHAVAN, S., AND STOLFO, S. Heisenbyte: Thwarting memory disclosure attacks using destructive code reads. In *Proc. of ACM SIGSAC* (2015), pp. 256–267.

[58] UBUNTU. Security/features - Ubuntu Wiki. https://wiki.ubuntu.com/Security/Features#Userspace_Hardening, 2016.

[59] WARTELL, R., MOHAN, V., HAMLEN, K. W., AND LIN, Z. Binary stirring: Self-randomizing instruction addresses of legacy x86 binary code. In *Proc. of ACM CCS* (2012), pp. 157–168.

[60] XU, J., KALBARCZYK, Z., AND IYER, R. Transparent runtime randomization for security. In *Proc. of IEEE SRDS* (2003), pp. 260–269.

[61] ZHANG, C., WEI, T., CHEN, Z., DUAN, L., SZEKERES, L., MCCAMANT, S., SONG, D., AND ZOU, W. Practical control flow integrity and randomization for binary executables. In *Proc. of IEEE SOSP* (2013).

[62] ZHANG, M., AND SEKAR, R. Control flow integrity for COTS binaries. In *Proc. of USENIX Security* (2013).

Don't Get Caught In the Cold, Warm-up Your JVM
Understand and Eliminate JVM Warm-up Overhead in Data-parallel Systems

David Lion, Adrian Chiu, Hailong Sun*, Xin Zhuang, Nikola Grcevski†, Ding Yuan
*University of Toronto, *Beihang University, †Vena Solutions*

Abstract

Many widely used, latency sensitive, data-parallel distributed systems, such as HDFS, Hive, and Spark choose to use the Java Virtual Machine (JVM), despite debate on the overhead of doing so. This paper analyzes the extent and causes of the JVM performance overhead in the above mentioned systems. Surprisingly, we find that the warm-up overhead, i.e., class loading and interpretation of bytecode, is frequently the bottleneck. For example, even an I/O intensive, 1GB read on HDFS spends 33% of its execution time in JVM warm-up, and Spark queries spend an average of 21 seconds in warm-up.

The findings on JVM warm-up overhead reveal a contradiction between the principle of parallelization, i.e., speeding up long running jobs by parallelizing them into short tasks, and amortizing JVM warm-up overhead through long tasks. We solve this problem by designing HotTub, a new JVM that amortizes the warm-up overhead over the lifetime of a cluster node instead of over a single job by reusing a pool of already warm JVMs across multiple applications. The speed-up is significant. For example, using HotTub results in up to 1.8X speed-ups for Spark queries, despite not adhering to the JVM specification in edge cases.

1 Introduction

A large number of data-parallel distributed systems are built on the Java Virtual Machine (JVM) [25]. These systems include distributed file systems such as HDFS [28], data analytic platforms such as Hadoop [27], Spark [64], Tez [62, 76], Hive [32, 77], Impala [13, 36], and key-value stores such as HBase [29] and Cassandra [15]. A recent trend is to process latency-sensitive, interactive queries [37, 65, 75] with these systems. For example, interactive query processing is one of the focuses for Spark SQL [10, 64, 65], Hive on Tez [37], and Impala [36].

Numerous improvements have been made to the performance of these systems. These works mostly focused on scheduling [2, 4, 31, 38, 56, 84], shuffling overhead [17, 19, 40, 45, 81], and removing redundant computations [61]. Performance characteristics studies [44, 46, 55, 57] and benchmarks [18, 23, 34, 80] have been used to guide the optimization efforts. Most recently, some studies analyzed the performance implications of the JVM's garbage collection (GC) on big data systems [24, 47, 48, 59].

However, there lacks an understanding of the JVM's overall performance implications, other than GC, in latency-sensitive data analytics workloads. Consequently, almost every discussion on the implications of the JVM's performance results in heated debate [35, 41, 42, 43, 58, 69, 83]. For example, the developers of Hypertable, an in-memory key-value store, use C++ because they believe that the JVM is inherently slow. They also think that Java is acceptable for Hadoop because "the bulk of the work performed is I/O" [35]. In addition, many believe that as long as the system "scales", i.e., parallelizes long jobs into short ones, the overhead of the JVM is not concerning [69]. It is clear that given its dynamic nature, the JVM's overhead heavily depends on the characteristics of the application. For example, whether an interpreted method is compiled to machine instructions by the just-in-time (JIT) compiler depends on how frequently it has been invoked. With all these different perspectives, a clear understanding of the JVM's performance when running these systems is needed.

This research asks a simple question: what is the performance overhead introduced by the JVM in latency-sensitive data-parallel systems? We answer this by presenting a thorough analysis of the JVM's performance behavior when running systems including HDFS, Hive on Tez, and Spark. We drove our study using representative workloads from recent benchmarks. We also had to carefully instrument the JVM and these applications to understand their performance. Surprisingly, after multiple iterations of instrumentation, we found that JVM warm-up time, i.e., time spent in class loading and interpreting bytecode, is a recurring overhead, which we made the focus of this study. Specifically, we made the following three major findings.

First, JVM warm-up overhead is significant even in I/O intensive workloads. We observed that queries from BigBench [23] spend an average of 21 seconds in warm-up time on Spark. Reading a 1GB file on HDFS from a hard drive spends 33% of its time in warm-up. We consider bytecode interpretation as an overhead because there is a huge performance discrepancy compared with JIT-compiled code (simply referred as *compiled code* in this paper) [39, 73]. For instance, we find that CRC checksum computation, which is one of the bottlenecks in HDFS read, is 230x faster when executed by compiled code rather than interpretation.

In addition, the warm-up time does not *scale*. Instead, it remains nearly constant. For example, the warm-up time in Spark queries remains at 21 seconds regardless of the workload scale factor, thus affecting short running jobs more. The broader implication is the following:

There is a contradiction between the principle of parallelization, i.e., speeding up long running jobs by parallelizing them into short tasks, and amortizing JVM warm-up overhead through long tasks.

Finally, the use of complex software stacks aggravates warm-up overhead. A Spark client loads 19,066 classes executing a query, which is 3 times more than Hive despite Spark's overall latency being shorter. These classes come from a variety of software components needed by Spark. In practice, applications using more classes also use more unique methods, which are initially interpreted. This results in increased interpretation time.

To solve the problem, our key observation is that the homogeneity of parallel data-processing jobs enables significant reuse rate of warm data, i.e., loaded classes and compiled code, when shared across different jobs. We designed HotTub, a new JVM that eliminates warm-up overhead by reusing JVMs from prior runs. It has the following advantages. First, it is a drop-in replacement of existing JVMs, abstracting away the JVM reuse, without needing users to modify their applications. In addition, it maintains *consistency* of an application's execution, i.e., the behavior is equivalent to the application being executed by an unmodified JVM except for the performance benefit [26]. Finally, it has a simple design that does not require a centralized component, and it selects the "best" JVM that will likely result in the highest re-usage of loaded classes and compiled code.

Evaluating HotTub shows that it can significantly speed-up latency sensitive queries. It reduces Spark's query latency on 100GB by up to 29 seconds, and speeds up HDFS reads on 1MB data by a factor of 30.08. In addition to warm-up time, the large speed up comes from more efficient use of cache, TLB, and branch predictor with up to 36% of miss rate reductions.

This paper makes the following contributions.

- It is the first analysis on the JVM's performance overhead in latency sensitive, data-parallel workloads. We are also the first to identify and quantify the warm-up overhead on such workloads.
- It presents HotTub, the first system that eliminates warm-up overhead while maintaining consistency.
- It also implements a set of improved JVM performance counters that measure the warm-up overhead. In particular, it is the first to provide fine-grained measurement of interpretation time.

The source code of HotTub and our JVM instrumentations in OpenJDK's HotSpot JVM are publicly available [1].

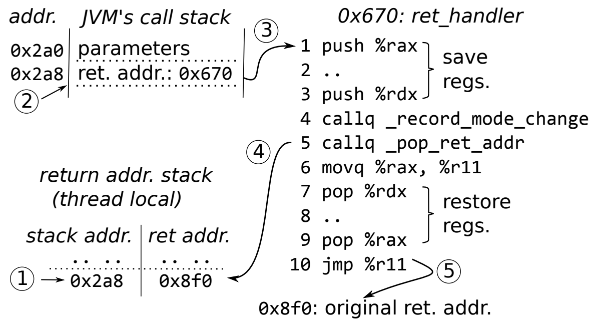
This paper has the following limitations. First, HotTub is less useful in long running workloads as the warm-up time is amortized by the long job completion time. In addition, HotTub does not completely comply with the Java Virtual Machine Specification [25] with regards to applications that use static variables whose initialization is timing dependent, which is rare and well-known to be a bad programming practice [1, 70].

This paper is organized as follows. Section 2 describes the instrumentations to the JVM used to measure its warm-up overhead. Section 3 presents the analysis of JVM performance. Section 4 and Section 5 describe HotTub's design, implementation, and limitations. We evaluate HotTub in Section 6. We survey the related work in Section 7 before we conclude.

2 Measure Warm-up Overhead

In this section we discuss how we instrument the JVM to measure its class loading and bytecode interpretation time with per-thread granularity. Section 3 describes how we use these instrumentations to study JVM overhead in data-parallel systems. We use OpenJDK's HotSpot

[1]`https://github.com/dsrg-uoft/hottub`



```
push %rax
..
push %rdx
callq _record_mode_change
callq _pop_ret_addr
movq %rax, %r11
pop %rdx
..
pop %rax
jmp %r11
```

Figure 1: Intercepting mode changing returns.

JVM, version 1.8.0 build 25.66. HotSpot is the primary reference Java Virtual Machine implementation [47].

Measuring per-thread class loading time is relatively straightforward. HotSpot already provides per-JVM class loading counters. We simply change the counter data structures to be thread local.

Measuring bytecode interpretation time is challenging. The JVM may be interpreting bytecode, executing JIT-compiled methods, or executing C/C++ compiled "native" code (referred as native execution). It requires us to instrument every *mode change*, i.e., transitions between interpreter execution and compiled/native execution. (We are not concerned with the transitions between compiled execution and native execution as our goal is to measure interpretation time.) If a mode change occurs via `call`, e.g., an interpreted method calls a compiled method or vice versa, it is straightforward to instrument, as it must first go through fixed program points known as adapters in HotSpot. Adapters are necessary because the interpreter uses a different stack layout from compiled or native execution. However, if a mode change occurs via `ret`, it is extremely difficult to instrument because the callee merely pops its stack frame, regardless of its caller. There is no one program point for `ret` that allows us to instrument the change back to the caller's mode.[2]

We instrument mode changing returns by replacing the original return address on the call stack with the address of our instrumented code. Figure 1 shows how it works in 5 steps: (1) when a mode changing `call` is executed, e.g., interpreter method A calls a compiled or native method B, we instrument this transition and also save the return address back to A (`0x8f0`) into a separate, thread local stack. (2) We replace the original return address with the address of our instrumented function `ret_handler` (`0x670`). (3) When B returns, it first jumps to `ret_handler`, which saves the registers that it is going to use. It records the mode change back to A, and pops the original return address (`0x8f0`) (step (4)). It then restores the saved registers, and in step (5) jumps to the original return address in A. `ret_handler` is implemented in 15 assembly instructions.

We have to carefully handle a few edge cases where the return address is used for special purposes. For GC, the JVM needs to walk each Java thread's stack to find live objects. Each frame's return address is used to identify its caller. Therefore we cannot leave the address of `ret_handler` on the stack; at the start of a GC pause, we restore the original return address. To quickly locate the original return address, we also save the address of the return address on the call stack (`0x2a8` in Figure 1). Similarly, the JVM uses the return address to propagate exceptions to caller methods. Therefore we restore the original return address upon throwing an exception.

The instrumentation incurs negligible overhead. When both class loading and interpreter counters are enabled on a range of HDFS workloads we used, the overhead is always less than 3.3%.

Note that class loading and interpreter times overlap, but our counter identifies this overlap. Therefore whenever we report JVM warm-up overhead as a single number, it is the sum of class loading and interpreter time subtracted by their overlap. However, we found only a small portion (14.8% in HDFS workload) of them overlap because class loading methods are quickly being JIT-compiled due to their frequent invocations.

3 Analysis of Warm-up Overhead

This section presents an in-depth analysis on JVM warm-up overhead in data-parallel systems. We first describe the systems used and our analysis method before presenting the analysis result. We also discuss existing industry practices that address the warm-up overhead and their limitations.

3.1 Methodology

We study HDFS, Hive running on Tez and YARN, and Spark SQL running with Spark in standalone mode. HDFS is a distributed file system. It is the default file system for many data parallel systems, including Spark and Hive. Both Spark and Hive process user queries by parallelizing them into short tasks, and are designed specifically for interactive queries [37, 65, 75]. They differ in how they parallelize the tasks: each Spark job runs

[2]The interpreter can also directly jump into compiled code via on-stack-replacement (OSR) [22, 33], a technique that immediately allows a hot loop body to run with compiled code during execution of the method. OSR also has to go through adapters, which we instrument, as the stack layout needs to be changed.

in only one JVM on each host (known as an *executor*), and utilizes multiple threads where each task runs in single thread (a JVM is a single process). In contrast, Hive on Tez runs each task in a separate JVM process known as YARN container. The versions we used are Hadoop-2.6.0, Spark-1.6.0, Hive-1.2.1, and Tez-0.7.0.

We benchmark Spark and Hive using BigBench [23]. It consists of 30 queries ranging over structured, semi-structured, and unstructured data that are modeled after real-world usage [23]. Its queries on structured data are selected from TPC-DS [79], which is widely used by SQL-on-Hadoop vendors like Cloudera [53], Hortonworks [54], Databricks [21, 66], and IBM [12] to drive their optimization efforts.

All experiments are performed on an in-house cluster with 10 servers. Four of them have 2 Xeon E5-2630V3, 16 virtual core, 2.4GHz CPUs with 256GB DDR4 RAM. The others have a single Xeon E5-2630V3 CPU with 128GB DDR4 RAM. Each server has two 7,200 RPM hard drives, is connected via 10Gbps interconnect, and runs Linux 3.16.0. The server components are long running and fully warmed-up for weeks and have serviced thousands of trial runs before measurement runs.

We run the queries on seven scale factors on Spark: 100, 300, 500, 700, 1K, 2K, 3K, and five scale factors on Hive: 100, 300, 500, 700, 1K. Each scale factor corresponds to the size of input data in GB (a scale factor 100 uses 100GB as input size, whereas 3K uses 3TB). For each scale factor, we repeat each query 10 times and take result from the fastest run in order to eliminate trials that might have been perturbed by other background workloads. In addition, we only analyze the 10 queries with the fastest job completion time out of the total 30 queries in BigBench, because of our focus on latency sensitive queries. (These queries are query 1, 9, 11, 12, 13, 14, 15, 17, 22, and 24.) BigBench is designed to be comprehensive, therefore many queries are long, batch processing queries instead of interactive queries. In addition, we found that at least 8 queries lead to heavy swapping at large data sizes, indicating that our system is not representative to run these queries.

We instrument each thread in the system with the per-thread class loading, interpreter, and GC performance counters to measure the JVM overhead of each parallel task. However, understanding the overall slow down of the entire job is non-trivial as the JVM overhead of multiple tasks can overlap. We borrow the blocked time analysis from Ousterhout *et al.* [55] to estimate the slow down to the entire job from per-task measurement. It works by first subtracting the time each task spends in the measured event (e.g., class loading) from its total execution time, and then simulates the scheduling of these tasks with the reduced execution time. We implemented a simple scheduling simulator with 500 LOC in Perl and simulated the original tasks with an accuracy of over 96%.

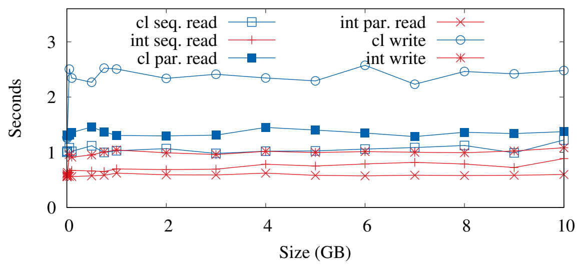


Figure 2: JVM warm-up time in various HDFS workload. "cl" and "int" represent class loading and interpretation time respectively. The x-axis shows the input file size.

Limitations. Our measurement of JVM overhead is a conservative underestimate. First, we do not measure the effect of the background threads that are used to JIT-compile bytecode. Similarly, we only measure the stop-the-world GC pause, ignoring background GC activities. This background work will compete with the application for CPU resources. In addition, our instrumentation may not cover all threads. For example, some libraries can create their own threads which we do not instrument. We use our best effort to address this problem: we instrumented the JVM thread constructor to observe the creation of every application thread, and instrument those that at least load classes. However, there are still threads that are not instrumented.

3.2 HDFS

We implement three different HDFS clients: sequential read, parallel read with 16 threads, and sequential write. We flush the OS buffer cache on all nodes before each measurement to ensure the workload is I/O bound. Note that interpreter time does not include I/O time because I/O is always performed by native libraries.

Figure 2 shows the class loading and interpreter time under different work loads. The average class loading times are 1.05, 1.55, and 2.21 seconds for sequential read, parallel read, and sequential write, while their average interpreter times are 0.74, 0.71, and 0.92 seconds. The warm-up time does not change significantly with different data sizes. The reason that HDFS write takes the JVM longer to warm-up is that it exercises a more complicated control path and requires more classes. Parallel read spends less time in the interpreter than sequential

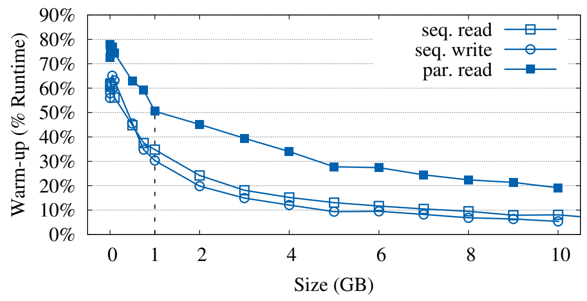


Figure 3: The JVM warm-up overhead in HDFS workloads measured as the percentage of overall job completion time.

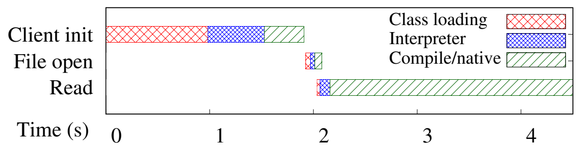


Figure 4: Breakdown of sequential HDFS read of 1GB file.

read because its parallelism allows the JVM to identify the "hot spot" faster.

Figure 3 further shows the significance of warm-up overhead within the entire job. Short running jobs are more significantly affected. When the data size is under 1GB, warm-up overhead accounts for more than 33%, 48%, and 30% of the client's total execution time in sequential read, parallel read, and sequential write. Sequential write suffers the least from warm-up overhead, despite its higher absolute warm-up time, because it has the longest run time. In contrast, parallel read suffers the most from warm-up overhead because of its short latency. According to a study [82] published by Cloudera, a vast majority of the real-world Hadoop workloads read and write less than 1GB per-job as they parallelize a big job into smaller ones. The study further shows that for some customers, over 60% of their jobs read less than 1MB from HDFS, whereas 1MB HDFS sequential read spends over 60% of its time in warm-up.

Next we break down class loading and interpreter time using the 1GB sequential read as an example. Figure 4 shows the warm-up time in the entire client read. A majority of the class loading and interpreter execution occurs before a client contacts a datanode to start reading.

Further drilling down, Figure 5 shows how warm-up time dwarfs the datanode's file I/O time. When the datanode first receives the read request, it sends a 13 bytes ack to the client, and immediately proceeds to send data packets of 64KB using the sendfile system call. The first

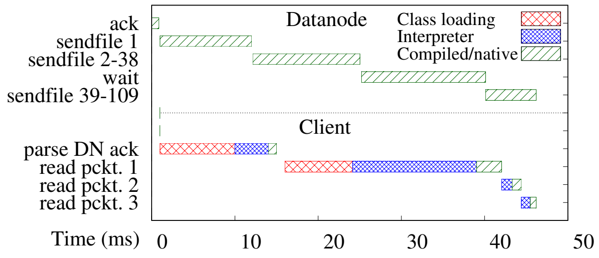


Figure 5: Breakdown of the processing of data packets by client and datanode.

	Read	Search	Define	Other	Total
Time (ms)	170	276	411	171	1,028

Table 1: Breakdown of class loading time.

sendfile takes noticeably longer than subsequent ones as the data is read from the hard drive. However, the client takes even longer (15ms) to process the ack because it is bottlenecked by warm-up time. By the time the client finishes parsing the ack, the datanode has already sent 11 data packets, thus the I/O time is not even on the critical path. The client takes another 26ms to read the first packet, where it again spends a majority of the time loading classes and interpreting the computation of the CRC checksum. By the time the client finishes processing the first three packets, the datanode has already sent 109 packets. In fact, the datanode is so fast that the Linux kernel buffer becomes full after the 38th packet, and it had to block for 14ms so that kernel can adaptively increase its buffer size. The client, on the other hand, is trying to catch up the entire time.

Figure 5 also shows the performance discrepancy between interpreter and compiled code. Interpreter takes 15ms to compute the CRC checksum of the first packet, whereas compiled code only takes 65μs per-packet.

Break down class loading. The HDFS sequential read takes a total of 1,028 ms to load 2,001 classes. Table 1 shows the breakdown of class loading time. Reading the class files from the hard drive only takes 170ms. Because Java loads classes on demand, loading 2,001 classes is broken into many small reads. 276ms are spent searching for classes on the classpath, which is a list of filesystem locations. The JVM specification requires the JVM to load the first class that appears in the classpath in the case of multiple classes with identical names. Therefore it has to search the classpath linearly when loading a class. Another 411ms is spent in define class, where the JVM parses a class from file into an in-memory data structure.

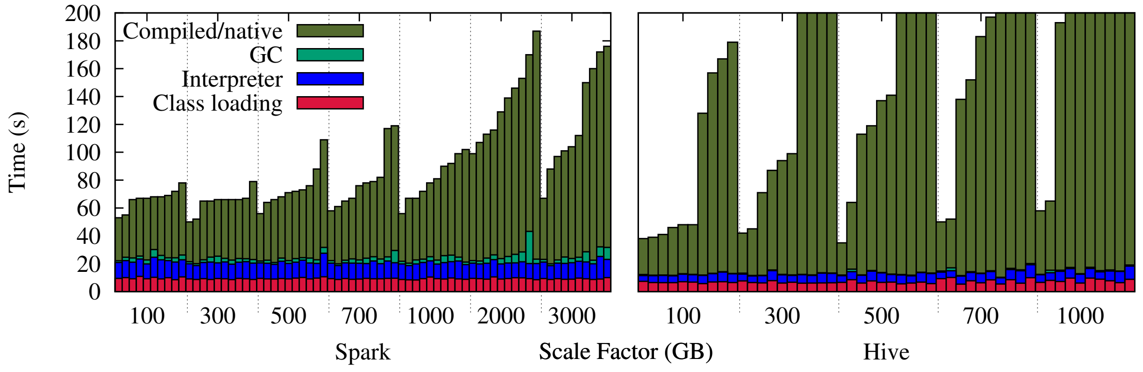


Figure 6: JVM overhead on BigBench. Overhead breakdown of BigBench queries across different scale factors. The queries are first grouped by scale factor and then ordered by runtime. Note that Hive has larger query time compared with Spark.

3.3 Spark versus Hive

Figure 6 shows the JVM overhead on Spark and Hive. Surprisingly, *each query spends an average of 21.0 and 12.6 seconds in warm-up time on Spark and Hive respectively*. Similar to HDFS, the warm-up time in both systems does not vary significantly when data size changes.

Software layers aggravate warm-up overhead. The difference in the warm-up times between Spark and Hive is explained by the difference in number of loaded classes. The Spark client loads an average of 19,066 classes, compared with Hive client's 5,855. Consequently, Spark client takes 6.3 seconds in class loading whereas the Hive client spends 3.7 seconds. A majority of the classes loaded by Spark client come from 10 third-party libraries, including Hadoop (3,088 classes), Scala (2,328 classes), and derby (1,110 classes). Only 3,329 of the loaded classes are from Spark packaged classes.

A large number of loaded classes also results in a large interpreter time. The more classes being loaded leads to an increase in the number of different methods that are invoked, where each method has to be interpreted at the beginning. On average, a Spark client invokes 242,291 unique methods, where 91% of them were never compiled by JIT-compiler. In comparison, a Hive client only invokes 113,944 unique methods, while 96% of them were never JIT-compiled.

Breaking down Spark's warm-up time. We further drill down into to one query (query 13 on SF 100) to understand the long warm-up time of Spark. While different queries exhibit different overall behaviors and different runtime, the pattern of JVM warm-up overhead is similar, as evidenced by the stable warm-up time. Figure 7 shows the breakdown of this query. The query completion time is 68 seconds, and 24.6 seconds are spent on warm-up overhead. 12.4 seconds of the warm-up time

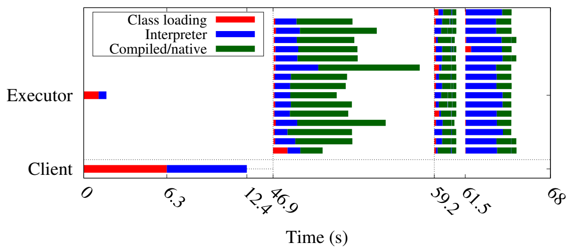


Figure 7: Breakdown of Spark's execution of query 13. It only shows one executor (there are a total of ten executors, one per host). Each horizontal row represents a thread. The executor uses multiple threads to process this query. Each thread is used to process three tasks from three different stages.

are spent on the client while the other 12.2 seconds come from the executors. Note that a majority of executors' class loading time is not on the critical path because executors are started immediately after the query is submitted, which allows executors' class loading time to be overlapped with the client's warm-up time. However, at the beginning of each stage the executor still suffers from significant warm-up overhead that comes primarily from interpreter time.

Hive. Hive parallelizes a query using different JVM processes, known as containers, whereas each container uses only one computation thread. Therefore within each container the warm-up overhead has a similar pattern with the HDFS client shown earlier. Hive and Tez also reuses containers to process tasks of the same query, therefore the JVM warm-up overhead can be amortized across the lifetime of a query.

3.4 Summary of Findings

Our analysis reveals that the JVM warm-up time is commonly the bottleneck of short running jobs, even when the job is I/O intensive. For example, 33% of the time in an HDFS 1GB sequential read is spent in warm-up, and 32% of the Spark query time on 100GB data size is on warm-up. The warm-up time stays nearly constant, indicating that its overhead becomes more significant in well parallelized short running jobs. In practice, many workloads are short running. For example, 90% of Facebook's analytics jobs have under 100GB input size [3, 9], and majority of the real-world Hadoop workloads read and write less than 1GB per-task [82]. Furthermore, Ousterhout *et al.* [56] show a trend of increasingly short running jobs with latency in the hundreds of milliseconds. This shows that both the data size and latency of data-parallel workloads are trending to be smaller. We also observe that multi-layered systems exacerbates the warm-up overhead, as they tend use more classes and methods, increasing class loading and interpretation times.

3.5 Industry Practices

While JVM performance has been actively studied over the last 20 years, most of the improvements focused on GC [24, 47, 48, 59] and JIT [39, 71, 73, 72, 74] instead of warm-up. One reason is that it is assumed that workloads are using long-running JVMs. For example, traditional JVM benchmarks, such as DayTrader [7] and SpecJBB/SpecJVM [67, 68], all assume the use of long running JVMs. This study has shown that this paradigm has changed on data-parallel systems, and efforts to address warm-up overhead should be increased moving forward.

Nevertheless, there exists some industry practices to address warm-up overhead. Despite the study showing clear overhead issues in Spark and Hive on Tez, both in fact already implement measures to reduce JVM warm-up overhead at the application layer. Both reuse the same JVM on each host to process the different tasks of each job (query in our case), thus amortizing the warm-up overhead across the life-time of a job. Spark runs one JVM or executor on each node that has the same life time as the job. Hive on Tez runs each task on a separate JVM (i.e., YARN container) and will try to keep reuse containers for new tasks. (Container reuse is the key feature introduced in Tez compared to Hadoop MapReduce.) A client could be designed to take multiple jobs from users and run them seemingly as one long job, which would allow multiple jobs to continue to use the same JVM.[3] However, it requires domain expertise to determine whether such reuse is safe. One must consider what static data should be re-initialized or which threads need to be killed. The use of third-party libraries further exacerbate the problem as they may contain stale static data and create additional threads. This is perhaps the reason that these systems do not allow JVM to be reused across different jobs unless the client is specifically designed to process multiple jobs. Nailgun [52] maintains a long-running JVM, and allows a client to dynamically run different applications on it. However, it does not take any measure to ensure that the reuse is consistent and the burden is on the users to decide whether a reuse is safe. In fact, naively reuse (unmodified) JVM does not even work when running the same Hive query twice, as the second run will crash because some static data in YARN needs to be properly reinitialized.

There are also a few solutions that change the JVM to address the warm-up overhead. The most advanced ones are perhaps on mobile platforms. The previous version of the Android runtime (ART) [6] (Android Marshmallow) would compile the entire app when it is first downloaded to gain native performance, but it suffered from the various limitations including large binary size and slow updates [5]. The latest version of ART (Nougat) [6] uses a new hybrid model. It first runs an app with an interpreter and JIT-compiles hot methods, similar to OpenJDK's HotSpot JVM. However, ART also stores profiling data after the app's run, allowing a background process to compile the select methods into native code guided by the profile. These methods now no longer suffer the warm-up overhead the next time this app is used. ART also statically initializes selected classes during the compilation and stores them in the application image to reduce class loading time.

The Excelsior JET [20], which is a proprietary JVM, compiles the bytecode statically into x86 native code before running the application, similar to older versions of ART. This eliminates both class loading and interpreted overhead, but this is at the cost of losing the performance benefit provided by profile-guided JIT compiler.

Other programming methods exist to reduce warm-up time. One can try to make JIT-compile more aggressively by changing the threshold with -XX:CompileThreshold. It is also possible to trigger class loading manually before classes are actually needed by either referencing the class or directly loading it. This is only useful if done off of the critical path. An example is that Spark's executor

[3]The container reuse in Tez is less predictable and cannot be taken advantage of by a smart user unlike with Spark, as there is a threshold for how long a container will be kept.

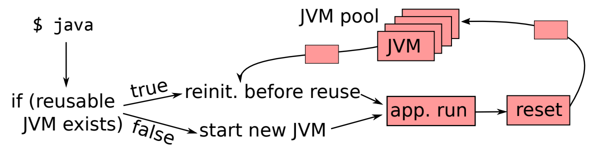


Figure 8: Architecture of HotTub.

is created before it actually receives any work allowing it to load classes. Similarly, one could potentially trigger JIT-compilation manually by invoking a method many times. Not only is this only useful if done off the critical path, but there are also other limitations. One has to ensure that the invocation has no side effects to the program state. Furthermore, one must also be wary of the parameters and path the method takes because the JIT-compiler is heavily guided by run-time profile, and unrealistic invocations could result in code less optimized for cases a developer cares about.

4 Design of HotTub

The design goal for HotTub is to allow applications to share the "warm" data, i.e., loaded classes and compiled code, thus eliminating the warm-up overhead from their executions. We considered two design choices: explicitly copy data among JVMs, or reuse the same JVMs after properly resetting states. We began implementation of the first design, trying to save class metadata and JIT-compiled code to disk for reuse in the next JVM process, similar to Android runtime [6]. We were able to share loaded classes, but eventually rejected this design because it is too complicated to maintain the consistency of all the pointers between the JVM address spaces. For example, the JIT-compiler does not produce relocatable code; a compiled method may directly jump to another compiled method. To maintain consistency, we either have to allocate all the loaded classes and compiled methods at the exact same addresses, which is inflexible, or fix all the pointer values, which is impractical as we have to interpret every memory address in compiled code. We chose the "reuse" design, which proved to be simpler, and we can leverage existing JVM features, such as garbage collection, to properly throw out stale data.

Figure 8 shows the architecture of HotTub. It is a drop-in replacement – users simply replace `java` with HotTub and can run their Java application with normal command. Running `java` will spawn a HotTub *client*, which attempts to contact a warmed-up JVM, known as a HotTub *server*, to run on. We refer to a reusable JVM as server because it is designed to be long running. After a server has completed a run, it will send the return

```
1  struct sockaddr_un add; // create unix sock.
2  char* sum = md5(classpath);
3  while (true) {
4    for (int i = 0; i < POOL_SIZE; i++) {
5     strcpy(add.sun_path,strcat(sum,itoc(i)));
6     if (connect(fd, add, sizeof(add))==0)
7       return reuse_server_and_wait(fd);
8     if (server_busy(i))
9       continue;
10    /* No JVM/server created. */
11    if (fork() == 0) // spawn new jvm in child
12      exec("/HotTub/java", args);
13    /* else, parent, go back to find server */
14   }
15 }
```

Figure 9: HotTub's client algorithm.

code to the client allowing the client to return normally to the user. The server will then run garbage collection, and reset the JVM state in preparation for the next client.

Next we discuss HotTub's client algorithm as shown in Figure 9. First, an important consideration is the reuse policy, i.e., which applications are allowed to share the same JVM. In order to gain the most benefit from an existing JVM it is ideal to run as similar a workload as possible on it. An application that performs similar logic and traverses the same code paths will reuse the same classes and compiled code. However, if the new application is significantly different from the previous one, then these benefits are reduced. In HotTub, a client first computes a checksum of the classpath and every file containing classes, which are generally JAR (Java Archive) files, on the classpath. Only the servers with the same checksum are candidates for reuse. While this limits reuse potential, it ensures large overlap of warm data. It also ensures that clients always uses the same classes, avoiding inconsistency problems.

In addition, the client appends an integer between 0 and POOL_SIZE to the checksum (line 5 in Figure 9), creating an ID to use as an address to contact a specific server. The client tries to connect to each ID in this range, and reuses the first connected server. If connect fails because the server is currently busy the client tries the next server. If connect fails because no server exists, or all servers are busy, the client forks a child process to create the server (line 11-12). The reason that we need to fork and exec java in the child, instead of directly exec java without fork, is that the user could be waiting for the java command to finish. Forking allows the parent to return after the application finishes, while the child process, which is now a warmed-up JVM, waits for reuse.

This design has a number of benefits. First, it is simple. The clients and servers on each node do not require a central coordinator, avoiding a potential bottleneck or central point of failure. In addition, it selects the longest running server that will likely result in the highest reusage of warm data. This is because the longest running server has had the most time to warm-up, JIT-compiling the most methods and loading the most classes. Finally, reusing the same JVM process across applications also offers caching benefits – between consecutive application runs the warm data stays in CPU caches because its memory address remains the same, and the OS does not need to flush the TLB.

4.1 Maintain Consistency

The main challenge to HotTub's design is to ensure that the application's execution on HotTub is *consistent* with the execution on an unmodified JVM. Data on stack and heap does not impose inconsistency problem, because at the end of a JVM application's execution, all of the application's stack frames have naturally ended. HotTub further garbage collects the heap objects with root references from the stack, therefore all of the heap objects that are application specific are also cleared. The remaining items, namely the loaded classes, compiled code, static variables, and file descriptors, need to be shared between reuse. Next we describe how HotTub maintains their consistency between reuse.

Class consistency. HotTub must ensure that any class it reuses is the same as what would be loaded by an unmodified JVM, as classes could potentially change in between runs, or during runs. Maintaining class consistency also ensures the consistency of compiled code as it is compiled from the class bytecode. The checksum mechanism used by the client only ensures the consistency of classes on the application classpath, which are loaded by the default class loader. While this accounts for the majority of loaded classes, an application can also implement a custom class loader, which has user-defined searching semantics, or dynamically generate a class.

Fortunately, any classes loaded by a custom class loader will not impose inconsistency issues for HotTub because a custom class loader must be instantiated by user code. This makes reuse impossible as every run will create a new instance of the class loader with no data from the previous run, causing it to load any class normally. Similarly, classes that are dynamically generated are loaded by custom class loaders in practice and are not an issue for consistency. However, there is no performance gained from reusing any classes that are loaded by custom class loaders as they are simply not reused.

Static variable consistency. At the end of application execution, static variables have values from the previous execution. Therefore HotTub needs to reinitialize them first to their default type value and then reinitialize them with their class initialization code. HotTub uses a simple policy. When the server is about to be reused, it reinitializes the static variables all at once by invoking the static initializer, namely `<clinit>`, of each class.

HotTub needs to maintain the correct order of the invocations to `<clinit>` of different classes. For example, class A's initialization may depend on class B having already been initialized. HotTub maintains the correct order by recording the order of class initializations when they are first initialized, and replaying the initializations in the same order before each reuse.

Unfortunately, reinitializing all the static data before the start of application is not consistent with the JVM specification [25] when the initialization of static variables have *timing* dependencies. Consider the following Java example:

```
Class A {
  static int a = 0;
  void foo () { a++; }
}
Class Bad { // bad practice
  static int b = A.a;
}
```

According to the JVM specification, the value of variable b in class Bad depends on when class A gets initialized. For example, if foo() has been called 3 times before Bad is referenced, then b will be initialized to 3. HotTub will initialize it to 0 in this case.

However, it is worth noting that static initialization that has timing depedency is a well known bad programming practices [1, 70]. It makes programs hard to reason about and difficult to test. Furthermore, multi-threading makes the matter worse as foo() in the previous example can be executed concurrently. In our experiments, we carefully examined the static initializers of the experimented systems, and none of them use such practice.

Another potential issue is when there exists a dependence cycle in the class initialization code of multiple classes, HotTub could lead to inconsistent behavior. For example, consider the following code snippet:

```
Class A {
  static int a = 10;
  static int b = B.b;
}
Class B {
  static int a = A.a; // set to 10 in
      HotSpot; 0 in HotTub
  static int b = 12;
}
```

There exists a circular dependency between class A and B in their static variables. Assume class A begins initialization first. Under HotSpot, it will first initialize A.a to 10 (line 2), and starts to initialize class B because A.b depends on it. When executing line 6, HotSpot detects that there is a circular dependency, and it will proceed to finish the initialization of the current class (class B), setting B.a to 10 and B.b to 12, before continuing the initialization of class A. HotTub, however, will run each static initializer from beginning to the end before moving on to the next one. Therefore in this case, it will first initialize class B because B's initialization finished before A's in the initial run. Since class A has not been initialized yet, HotTub will set B.a to the initial value of A.a, which is 0, leading to an inconsistent value on B.a. Note that circular dependence in static initializers is also a known bad practice, and the JVM specification explicitly warns about its dangers and discourages its use [25].

File descriptor consistency. HotTub will close the file descriptors (fd) opened by the application at the reset phase so that they will not affect the next client run. The only remaining open fds are those opened by the JVM itself, mostly for JAR files. HotTub also closes stdin, stdout, and stderr at the end of an application's execution in the reset stage. After the client selects a server JVM for reuse, the client first sends all fds it has opened to the server, including stdin, stdout, and stderr, so that the server can inherit these file descriptors and have the same same open files as the client.

However, it is possible that a file descriptor opened by the client conflicts with an open file descriptor used by the server JVM. For example, if the user invokes HotTub with the command `$ java 4>file`, HotTub cannot reuse a server with fd 4 in use. Therefore when selecting a server for reuse, HotTub also checks if the server has open fds that conflict with a client's redirected fd, and only reuse servers that do not have such a conflict.

Handling signals and explicit exit. HotTub has to handle signals such as SIGTERM and SIGINT and explicit exit by the application, otherwise it will lose the target server process from our pool. If application registers its own signal handler, HotTub forwards the signal. Otherwise, HotTub handles signals and application exits by unwinding the stack of non-daemon Java threads and killing them. Java "daemon" threads are not normally cleaned at JVM exit as they simply exit with the process. However, for consistency, HotTub must kill these threads. This sets the JVM to the same state as if the application finishes normally. The server then closes connection to client, so the client exits normally. However, if the application calls _exit in a native library, HotTub cannot save this server process from being terminated.

4.2 Limitations

HotTub cannot handle SIGKILL. Therefore, if the user sends `kill -9` to a HotTub server we will lose it for future reuse. However, it is most likely that the user only wants to kill the client `java` process, which will not cause us to lose the server because the server and client are in different processes.

Unfortunately, this use of separate processes can raise problems if a user expects the application to run in the same process as `java`. For example, YARN terminates a container by first sending SIGTERM to the process identified by a PID file, followed by SIGKILL. This would not cause HotTub to violate consistency, as the server will be killed and the client subsequently exits on a closed connection. However, this will disable HotTub from reusing the server. Therefore we had to modify the management logic in YARN to disable "kill -9".

The use of HotTub raises privacy concerns. HotTub limits reuse to the same Linux user, as cross user reuse allows a different user to execute code with the privileges of the first user. However, our design still violates the principle "base the protection mechanisms on permission rather than exclusion" [63]. Although we carefully clear and reset data from the prior run, an attacker could still reconstruct the partial execution path of the prior run via timing channel. For example, by measuring the execution time of the first time invocation of a method the attacker can infer whether this method has been executed, and thus JIT-compiled, in the prior run. In our current implementation we are not zeroing out the heap space after GC. This allows malicious users to use native libraries to read the heap data from prior runs.

HotTub cannot maintain consistency if the application rewrites the bytecode or compiled code of a class on the classpath after it has been loaded, and does not write the modifications back to the class file. In such cases, the in-memory bytecode or compiled code will be different from the checksum computed by HotTub. It is difficult to detect the bytecode rewriting because the application can always bypass the JVM using a native library to modify any memory locations in the address space. However, modifying the bytecode of a loaded class is undefined behavior as the JVM may be already using a compiled method, thus the changes to bytecode will have no effect. In practice we have never encountered such cases. Note that the HotSpot JVM performs its own form of bytecode rewriting, which is not a problem for HotTub as this is only done for performance optimizations and preserves the original semantic.

HotTub currently only targets the Java Virtual Machine runtime. Other runtimes such as Microsoft's Common Runtime Language (CLR) [51] also exhibits similar warm-up overhead properties. Similar to class loading done by the JVM, CLR must load portable executable (PE) file, which is similar to a Java classfile, that contains metadata required at runtime such as type definitions and member signatures. To the best of our knowledge CLR operates similar to typical JVMs, where an interpreter will execute bytecode or an intermediate language, until a JIT-compiler can produce native code for the method. Having these properties should exhibit similar warm-up overhead of class loading and interpretation in CLR, so implementing HotTub for CLR could potentially produce similar speed-ups.

5 Implementation of HotTub

The client is implemented as a stand-alone program with 800 lines of C code, and the server is implemented on top of OpenJDK's HotSpot JVM by adding approximately 800 lines of C/C++ code. We use Unix domain sockets to connect a client with servers. A nice feature of Unix domain socket is that it allows processes to send file descriptors. Therefore the client simply send its open file descriptors, including stdin, stdout, stderr together with other redirected ones, to the server. This avoids sending the actual input and output data across processes. Next we discuss the implementation details of HotTub.

Threads management. HotTub does not use any additional thread in JVM for its management task. Instead, it uses the Java main thread. At the end of the application execution after the `main()` method finishes, we do not terminate the Java main thread. Instead we uses it to perform the various reset tasks including (1) kill other Java threads, (2) set all static variables to their default type value, (3) garbage collect the heap, (4) optionally unload native libraries.[4] It then waits for a client connection. When it receives another client connection, it reinitializes the static variables, sets up the file descriptors properly, sets any Java properties, sets any environment variables, and finally invokes the `main()` method.

Complication arises when the JVM receives a signal or a thread calls `System.exit`. In these cases, we need to use the thread that receives the signal or calls `System.exit` to clean up the all Java threads.

Static reinitialization. HotTub handles a few technical challenges when implementing the replay of class initialization. One challenge is with enumeration classes. For each class in Java there exists a `java.lang.Class` object that contains information about the class, such as the methods and fields it contains, so that it can be queried for reflection inspections. For enumeration classes, this object contains a mapping of each enumeration constant string name to its object. Because after reinitialization there will be new objects allocated for each enumeration constant, HotTub also has to update the mapping in this `java.lang.Class` object in each class.

Another challenge is the JIT-compiler's inlining of static final references, i.e., a compiled method could directly reference the address of a static final object. However, after reinitialization, a new object will be created so that the old reference is no longer valid. HotTub solves this by disabling the inlining of static final references.

[4]Theoretically we should unload native libraries because HotTub does not verify their consistencies. However we observe that native libraries typically do not impose inconsistency issues, e.g., they typically do not use static data. Therefore we make unloading them optional.

Completion time (s)	Unmod.	HotTub	Speed-up
HDFS read 1MB	2.29	0.08	30.08x
HDFS read 10MB	2.65	0.14	18.04x
HDFS read 100MB	2.33	0.41	5.71x
HDFS read 1GB	7.08	4.26	1.66x
Spark 100GB best	65.2	36.2	1.80x
Spark 100GB median	57.8	35.2	1.64x
Spark 100GB worst	74.8	54.4	1.36x
Spark 3TB best	66.4	41.4	1.60x
Spark 3TB median	98.4	73.6	1.34x
Spark 3TB worst	381.2	330.0	1.16x
Hive 100GB best	29.0	16.2	1.79x
Hive 100GB median	38.4	25.0	1.54x
Hive 100GB worst	206.6	188.4	1.10x

Table 2: Performance improvements by comparing the average job completion time of an unmodified JVM and HotTub. For Spark and Hive we report the average times of the queries with the, best, median, and worst speed-up for each data size.

6 Performance of HotTub

We conduct a variety of experiments on HotTub to evaluate its performance on the following dimensions: (1) speed-up over an unmodified JVM repeating the same workload; (2) speed-up when running different workloads; (3) management overhead imposed by HotTub (e.g., reset, client/server management). All experiments are performed on the same environment and settings as described in Section 3.

6.1 Speed-up

Table 2 shows HotTub's speed-up compared with unmodified HotSpot JVM. We run the same workload five times on an unmodified JVM and six times on HotTub.

Perf. counter	Executor						Client					
	U	H	U/H	U Rate	H Rate	Rate Diff.	U	H	U/H	U Rate	H Rate	Rate Diff.
L1-dcache-misses	171M	81M	2.1x	1.839%	1.994%	-8.416%	154M	21M	7.3x	6.254%	6.115%	2.218%
L1-icache-misses	40M	13M	3.1x	-	-	-	44M	6M	7.3x	-	-	-
page faults	543K	122K	4.4x	-	-	-	851K	227K	3.7x	-	-	-
dTLB-load-misses	4,431M	3,087M	1.4x	0.080%	0.051%	36.418%	2,999M	375M	8.0x	0.327%	0.295%	9.894%
iTLB-load-misses	704M	228M	3.1x	3.424%	3.294%	3.805%	755M	97M	7.8x	3.359%	3.054%	9.078%
branch-misses	1,158M	597M	1.9x	0.913%	0.646%	29.234%	974M	119M	8.2x	3.270%	2.971%	9.141%

Table 3: Comparing cache, TLB, and branch misses between HotTub (H) and an unmodified JVM (U) when running query 11 of BigBench on Spark with 100GB data size. The numbers are taken from the average of five runs. All page faults are minor faults. "Rate diff." is calculated as (U Rate - H Rate)/(U Rate), which shows the improvement of HotTub on the miss rate. Perf cannot report the number of L1-icache loads or memory references to know the corresponding rates.

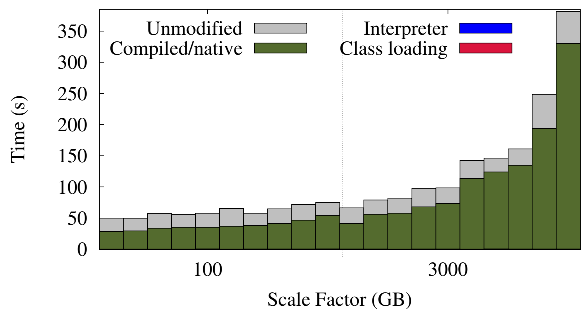


Figure 10: HotTub successfully eliminates warm-up overhead. Unmodified query runtime shown against a breakdown of a query with reuse. There are 10 queries run on 2 scale factors for BigBench on Spark. Interpreter and class loading overhead are so low they are unnoticeable making up the difference.

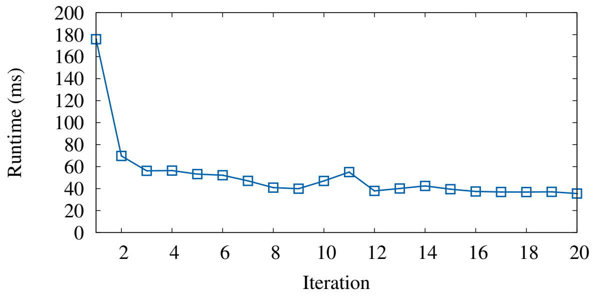


Figure 11: HotTub iterative runtime improvement. A Sequential 1MB HDFS read performed repeatedly by HotTub. Iteration 0 is the application runtime of a new JVM, while iteration N is the Nth reuse of this JVM.

We compare the average runtime of the five unmodified runs with the five reuse HotTub runs, excluding the initial warm-up run. For Spark and Hive, we run the same 10 queries that we used in our study. Note that in this experiment the systems we run are unmodified, unlike the ones we used in Section 3 that had to be instrumented. Therefore the unmodified systems' runtimes are slightly faster than the ones in Section 3.

The results shows that HotTub significantly speeds up the total execution time. For example, HotTub reduces the average job completion time of the Spark query with the highest speed-up by 29 seconds on 100GB data, and can speed-up HDFS 1MB read by a factor of 30.08. Amongst nearly 200 pairs of trials, a job running in a reused HotTub JVM always completed faster compared to an unmodified JVM.

Enabling our performance counters, we observe that indeed HotTub eliminates the warm-up overhead. In all the experiments, the server JVM spends less than 1% of the execution time in class loading and interpreter. Figure 10 shows Spark queries running on HotTub with nearly zero warm-up overhead.

Figure 11 shows how the runtime decreases over the number of reuses. While the significant speedup comes from the first time reuse, it also shows that for this particular short running job the JVM will not be completely warm by the end of the first run and require multiple iterations before reaching peak performance. Figure 11 also shows that it takes 12 iterations before the JVM becomes fully warmed-up. This further suggests that short running jobs cannot even reach max JVM performance by the end of its execution, which further emphasizes the necessity of reusing JVMs on short jobs. Long running jobs, however, will likely fully warm up the JVM before their execution ends on first run.

To understand HotTub's performance behavior in detail, we further compare the hardware performance counters. Table 3 shows the result. HotTub already significantly reduces the number of memory accesses because the classes are already loaded and bytecode compiled. For the Spark executor there are almost half as many cache references reported by perf and the Spark client shows an even higher reduction, up to 9x. The reduction in accesses appears large, but is consistent with the 1.74x speed-up experienced by running this query on HotTub.

		Testing				
		q11	q14	q15	q09	q01
Training	q11	1.78	1.67	1.51	1.49	1.55
	q14	1.64	1.65	1.47	1.49	1.50
	q15	1.72	1.67	1.62	1.54	1.62
	q09	1.57	1.59	1.55	1.53	1.53
	q01	1.76	1.74	1.65	1.54	1.74

Table 4: Speed-up sensitivity to workload differences using the five fastest BigBench queries on Spark with 100GB data.

It further reduces the number of various cache misses. The reduction comes from multiple sources. First, HotTub results in less memory accesses and smaller footprint because some data is no longer needed to be accessed (e.g., bytecode that are already compiled will not be accessed). Second, because the applications run in the same JVM process, warm data does not need to be reallocated between the runs, therefore the cached data can be reused and the number of cold misses get reduced. The OS also does not need to flush the TLB between the runs. Finally, the reduction of instruction cache and iTLB misses is likely afforded by eliminating interpreter execution and JIT-compiler's instruction cache usage optimization. For example, JIT-compiler will arrange the basic blocks in the order of frequently taken branches.

The numbers in Table 3 also show that HotTub reduces the accesses and misses in the Spark client much more than the executor. This is likely due to the nature of the work each component does. The executor is processing large amounts of data, taking the majority of time and memory references, even in a reused run, while the client performs much less work. Since the warm-up overhead reduction is constant, it follows that the executor should be less affected, while the client will be heavily affected as it spent more of its time performing warm-up.

6.2 Sensitivity to Difference in Workload

Table 4 compares HotTub's performance sensitivity to the workload differences between training and testing runs. We warm up the JVM by repeatedly run a single "training query" four times, and apply it once on the "testing query". Note that we cannot apply the test run more than once for this experiment because the JVM will then be warming up with the testing query. We repeat this process five times and take the average runtime of the testing queries, and then report the speed-up of this average runtime over the runtime of unmodified JVM running the testing query. The result shows that, for our tests, HotTub can achieve at least 1.47 speed-up.

Query 11 and 1 observe the largest speed-up when the training and testing queries are the same. The other queries observe best speed-up when running on a JVM trained from a different query. This is due to the large variance observed in our experiment. All of the measured runtime of testing queries fall into the range of $(mean - variance, mean + variance)$, where mean and variance are of the five measured runs where the training and testing queries are the same. This also indicates that different queries use many similar classes and code.

6.3 Management Overhead

Compared with an unmodified JVM, HotTub adds overhead in three phases: when a client connects to a server, when the server runs class initialization, and when the server resets the static data. The first two are on the critical path of the application latency while the third merely affects how early this JVM can be reused again. The overhead for connecting to a server when there are no servers in the pool is 81ms. Once servers are available for re-use, the connection overhead drops to $300\mu s$. The overhead added to the critical path from class reinitialization is, on average, 350ms for Hive on Tez containers, 400ms for Spark executors, and 720ms for Spark clients. The time taken to reset static data is dominated by garbage collection and only takes no more than 200ms because the application's stack frames have ended, thus there are few roots from which GC has to search from. Root objects are objects assumed to be reachable, such as static variables or variables in live stack frames.

HotTub also adds overhead on the memory usage of the system as the server processes remain alive after the application finishes. The number of servers to be left alive on a node can be configured by the user, for our evaluation we arbitrarily chose 5. An unused server takes up approximately 1GB of memory.

7 Related Work

We discuss prior studies on the performance of the JVM and data-parallel distributed systems. Commercial and industry solutions have been discussed in Section 3.5. Our work distinguishes itself as it the first to study the performance implications of JVM warm-up overhead on data-parallel systems.

Performance of garbage collection (GC). Recently, a few proposals are made to improve the GC performance for big data systems [24, 47, 48, 59]. Gog *et al.* observe that objects often have clearly defined lifetimes in big data analytics systems [24]. Therefore they propose a region-based memory management [78] system where developers can annotate the objects with the same lifetime. Maas *et al.* [47, 48] observe that different JVMs running the same job often pause to garbage collect at the same time given the homogeneous nature of their executions, therefore they propose a system named Taurus that coordinates the GC pauses among different JVM processes. Our work is complementary as we focus on studying the JVM warm-up overhead, while Broom and Taurus only focused on GC. HotTub can also be integrated together with Broom and Taurs to provide comprehensive speed-up of the JVM runtime. Comparing the design of Taurus and HotTub also reveals interesting trade-offs. Taurus does not modify the JVM itself, therefore users will have less reliability concerns in deployment. However, its capability to control the JVM is restricted to the interfaces JVM exposed. Taurus requires the JVMs in the network to coordinate via a consensus protocol, whereas HotTub uses a simpler design that makes it standalone and does not require network communication. Consequently HotTub can also benefit non-distributed applications.

Other papers studied GC performance on non-distributed workload. Appel [8] uses theoretical analysis to argue that when physical memory is abundant, GC can achieve high performance comparable to other memory management techniques. Hertz *et al.* further validated this via more thorough experimental evaluation [30], but they also found that the performance of GC deteriorate quickly when free memory becomes scarce. Others have compared the performance of general GC algorithms (e.g., generational GC) versus customized ones and concluded that general algorithms achieve good performance in most cases [11, 14, 85].

Performance studies on data-parallel systems. A handful of works have thoroughly analyzed the performance of data-parallel systems [44, 46, 55, 57]. However they did not study the JVM performance impact. Ousterhout *et al.* comprehensively studied the performance of Spark [55], and revealed that network and disk I/O are no longer the bottleneck. Interestingly, they found that CPU is often the bottleneck, and a large amount of CPU time is spent in data deserialization and decompression. However, because they only analyzed Spark itself, they did not further drill down to provide a low level understanding of such high CPU time. Using the same workload, our study suggests that the class loading and bytecode interpretation are likely the main cause of deserialization and decompression. Pavlo *et al.* [57] compared Hadoop MapReduce with DBMS, and found that data shuffling is often the bottleneck for MapReduce. Jiang *et al.* [44] analyzed the performance of Hadoop MapReduce. McSherry *et al.* [50] surveyed the existing literature on data-parallel systems and their experimental workload, and concluded that many of the workloads use small data input sizes that can be well handled by a single threaded implementation. This has similar implications as the other studies on real-world analytic jobs where most jobs are short running because of the small input size [82], where the JVM warm-up time is even more significant.

Other works on improving data-parallel systems performance focused on scheduling [2, 4, 31, 38, 56, 84], high performance interconnect [17, 19, 40, 45, 81], optimization for multi-cores [16, 49, 60], and removing redundant operations [61]. Our work is complementary as it focuses on JVM-level improvements.

8 Concluding Remarks

We started this project with the curiosity to understand the JVM's overhead on data-parallel systems, driven by the observation that systems software is increasingly built on top of it. Enabled by non-trivial JVM instrumentations, we observed the warm-up overhead, and were surprised by the extent of the problem. We then pivoted our focus on to the warm-up overhead by first presenting an in-depth analysis on three real-world systems. Our result shows the warm-up overhead is significant, and can be exacerbated as jobs become more parallelized and short running. We further designed HotTub, a drop-in replacement of the JVM that can eliminate warm-up overhead by amortizing it over the lifetime of a host. Evaluation shows it can speed-up systems like HDFS, Hive, and Spark, with a best case speed-up of over 30.08X.

Acknowledgements

We greatly appreciate the insightful feedbacks from the anonymous reviewers and our shepherd Andrea Arpaci-Dusseau. We thank Yu Luo, Serhei Makarov, Michael Stumm, Jenny Ren, Kirk Rodrigues, Guangji Xu, Yongle Zhang, and Xu Zhao for the useful and thought stimulating discussions. We thank Yu Luo for setting up and maintaining the server cluster environment used in our experiments. His help was invaluable. This research is supported by NSERC Discovery grant, NetApp Faculty Fellowship, and an NSERC USRA award. Hailong Sun is supported by National Key Research and Development Program of China (2016YFB1000804) and National Natural Science Foundation of China (61370057).

References

[1] A case against static initializers. `http://sensualjava.blogspot.com/2008/12/case-against-static-initializers.html`.

[2] F. Ahmad, S. T. Chakradhar, A. Raghunathan, and T. N. Vijaykumar. Tarazu: Optimizing mapreduce on heterogeneous clusters. In *Proceedings of the Seventeenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XVII, 2012.

[3] G. Ananthanarayanan, A. Ghodsi, A. Wang, D. Borthakur, S. Kandula, S. Shenker, and I. Stoica. Pacman: Coordinated memory caching for parallel jobs. In *Proceedings of the 9th USENIX Conference on Networked Systems Design and Implementation*, NSDI'12, 2012.

[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 *Proceedings of the 9th USENIX Conference on Operating Systems Design and Implementation*, OSDI '10, 2010.

[5] Android ART Just-In-Time (JIT) Compiler. `https://source.android.com/devices/tech/dalvik/jit-compiler.html`.

[6] Android runtime (ART). `https://source.android.com/devices/tech/dalvik/index.html`.

[7] Apache Geronimo DayTrader Benchmark. `http://geronimo.apache.org/GMOxDOC20/daytrader.html`.

[8] A. W. Appel. Garbage collection can be faster than stack allocation. *Inf. Process. Lett.*, 25(4).

[9] R. Appuswamy, C. Gkantsidis, D. Narayanan, O. Hodson, and A. Rowstron. Scale-up vs scale-out for hadoop: Time to rethink? In *Proceedings of the 4th Annual Symposium on Cloud Computing*, SOCC '13, 2013.

[10] M. Armbrust, R. S. Xin, C. Lian, Y. Huai, D. Liu, J. K. Bradley, X. Meng, T. Kaftan, M. J. Franklin, A. Ghodsi, and M. Zaharia. Spark SQL: Relational data processing in spark. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, SIGMOD '15, 2015.

[11] E. D. Berger, B. G. Zorn, and K. S. McKinley. Reconsidering custom memory allocation. In *Proceedings of the 17th ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications*, OOPSLA '02, 2002.

[12] Big SQL 3.0: Hadoop-DS benchmark-Performance isn't everything. `https://developer.ibm.com/hadoop/blog/2014/12/02/big-sql-3-0-hadoop-ds-benchmark-performance-isnt-everything/`.

[13] M. K. A. B. V. Bittorf, T. Bobrovytsky, C. C. A. C. J. Erickson, M. G. D. Hecht, M. J. I. J. L. Kuff, D. K. A. Leblang, N. L. I. P. H. Robinson, D. R. S. Rus, J. R. D. T. S. Wanderman, and M. M. Yoder. Impala: A modern, open-source sql engine for hadoop. In *Proceedings of the 7th Biennial Conference on Innovative Data Systems Research*, CIDR '15, 2015.

[14] S. M. Blackburn, P. Cheng, and K. S. McKinley. Myths and realities: The performance impact of garbage collection. In *Proceedings of the Joint International Conference on Measurement and Modeling of Computer Systems*, SIGMETRICS '04/Performance '04, 2004.

[15] Cassandra. `http://cassandra.apache.org`.

[16] R. Chen, H. Chen, and B. Zang. Tiled-mapreduce: Optimizing resource usages of data-parallel applications on multicore with tiling. In *Proceedings of the 19th International Conference on Parallel Architectures and Compilation Techniques*, PACT '10, 2010.

[17] T. Condie, N. Conway, P. Alvaro, J. M. Hellerstein, K. Elmeleegy, and R. Sears. Mapreduce online. In *Proceedings of the 7th USENIX Conference on Networked Systems Design and Implementation*, NSDI '10, 2010.

[18] B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears. Benchmarking cloud serving systems with YCSB. In *Proceedings of the 1st ACM Symposium on Cloud Computing*, SoCC '10, 2010.

[19] P. Costa, A. Donnelly, A. Rowstron, and G. O'Shea. Camdoop: Exploiting in-network aggregation for big data applications. In *Proceedings of the 9th USENIX Conference on Networked Systems Design and Implementation*, NSDI '12, 2012.

[20] Excelsior JET - Java Virtual Machine (JVM) and Native Code Compiler. `https://www.excelsiorjet.com/`.

[21] Exciting performance improvements on the horizon for spark sql. `https://databricks.com/blog/2014/06/02/exciting-performance-improvements-on-the-horizon-for-spark-sql.html`.

[22] S. J. Fink and F. Qian. Design, implementation and evaluation of adaptive recompilation with on-stack replacement. In *Proceedings of the International Symposium on Code Generation and Optimization: Feedback-directed and Runtime Optimization*, CGO '03, 2003.

[23] 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, 2013.

[24] I. Gog, J. Giceva, M. Schwarzkopf, K. Vaswani, D. Vytiniotis, G. Ramalingam, M. Costa, D. G. Murray, S. Hand,

and M. Isard. Broom: Sweeping out garbage collection from big data systems. In *15th Workshop on Hot Topics in Operating Systems*, HotOS '15, 2015.

[25] J. Gosling, B. Joy, G. Steele, G. Bracha, and A. Buckley. The Java®Virtual Machine specification - Java SE 8 Edition. `https://docs.oracle.com/javase/specs/jvms/se8/html/`.

[26] C. Gray and D. Cheriton. Leases: An efficient fault-tolerant mechanism for distributed file cache consistency. In *Proceedings of the Twelfth ACM Symposium on Operating Systems Principles*, SOSP '89, 1989.

[27] Hadoop. `https://hadoop.apache.org`.

[28] Hadoop Distributed File System (HDFS). `http://hadoop.apache.org/docs/stable/hdfs_design.html`.

[29] Hbase. `http://hbase.apache.org/`.

[30] M. Hertz and E. D. Berger. Quantifying the performance of garbage collection vs. explicit memory management. In *Proceedings of the 20th Annual ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications*, OOPSLA '05, 2005.

[31] 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, 2011.

[32] Hive. `http://hive.apache.org`.

[33] U. Hölzle and D. Ungar. A third-generation self implementation: Reconciling responsiveness with performance. In *Proceedings of the Ninth Annual Conference on Object-oriented Programming Systems, Language, and Applications*, OOPSLA '94, 1994.

[34] S. Huang, J. Huang, Y. Liu, L. Yi, and J. Dai. Hibench: A representative and comprehensive hadoop benchmark suite. In *Proc. ICDE Workshops*, 2010.

[35] hypertable: why we chose CPP over Java. `https://code.google.com/p/hypertable/wiki/WhyWeChoseCppOverJava`.

[36] Impala – Cloudera. `http://www.cloudera.com/content/www/en-us/products/apache-hadoop/impala.html`.

[37] Interactive query with apache hive on apache tez. `http://hortonworks.com/hadoop-tutorial/supercharging-interactive-queries-hive-tez/`.

[38] M. Isard, V. Prabhakaran, J. Currey, U. Wieder, K. Talwar, and A. Goldberg. Quincy: Fair scheduling for distributed computing clusters. In *Proceedings of the ACM SIGOPS 22Nd Symposium on Operating Systems Principles*, SOSP '09, 2009.

[39] K. Ishizaki, M. Takeuchi, K. Kawachiya, T. Suganuma, O. Gohda, T. Inagaki, A. Koseki, K. Ogata, M. Kawahito, T. Yasue, T. Ogasawara, T. Onodera, H. Komatsu, and T. Nakatani. Effectiveness of cross-platform optimizations for a java just-in-time compiler. In *Proceedings of the 18th Annual ACM SIGPLAN Conference on Object-oriented Programing, Systems, Languages, and Applications*, OOPSLA '03, 2003.

[40] N. S. Islam, M. W. Rahman, J. Jose, R. Rajachandrasekar, H. Wang, H. Subramoni, C. Murthy, and D. K. Panda. High performance rdma-based design of hdfs over infiniband. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, SC '12, 2012.

[41] Quora: In what cases is Java faster than C. `https://www.quora.com/In-what-cases-is-Java-faster-if-at-all-than-C`.

[42] Quora: In what cases is Java slower than C by a big margin. `https://www.quora.com/In-what-cases-is-Java-slower-than-C-by-a-big-margin`.

[43] StackOverflow: Why do people still say Java is slow? `http://programmers.stackexchange.com/questions/368/why-do-people-still-say-java-is-slow`.

[44] D. Jiang, B. C. Ooi, L. Shi, and S. Wu. The performance of mapreduce: An in-depth study. *Proc. VLDB Endow.*, 3(1-2).

[45] MapReduce-4049: Plugin for generic shuffle service. `https://issues.apache.org/jira/browse/MAPREDUCE-4049`.

[46] K.-H. Lee, Y.-J. Lee, H. Choi, Y. D. Chung, and B. Moon. Parallel data processing with mapreduce: A survey. *SIGMOD Rec.*, 40(4).

[47] M. Maas, K. Asanović, T. Harris, and J. Kubiatowicz. Taurus: A Holistic Language Runtime System for Coordinating Distributed Managed-Language Applications. In *Proceedings of the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '16, 2016.

[48] M. Maas, T. Harris, K. Asanović, and J. Kubiatowicz. Trash day: Coordinating garbage collection in distributed systems. In *15th Workshop on Hot Topics in Operating Systems*, HotOS '15, 2015.

[49] Y. Mao, R. Morris, and M. F. Kaashoek. Optimizing mapreduce for multicore architectures. Technical report, Massachusetts Institute of Technology, 2010.

[50] F. McSherry, M. Isard, and D. G. Murray. Scalability! But at what COST? In *15th Workshop on Hot Topics in Operating Systems*, HotOS '15, 2015.

[51] Microsoft Common Language Runtime (CLR). `https://msdn.microsoft.com/en-us/library/8bs2ecf4(v=vs.110).aspx`.

[52] Nailgun: Insanely fast Java. http://www.martiansoftware.com/nailgun/background.html.

[53] New benchmarks for sql-on-hadoop: Impala 1.4 widens the performance gap. http://blog.cloudera.com/blog/2014/09/new-benchmarks-for-sql-on-hadoop-impala-1-4-widens-the-performance-gap/.

[54] New benchmarks for sql-on-hadoop: Impala 1.4 widens the performance gap. http://blog.cloudera.com/blog/2014/09/new-benchmarks-for-sql-on-hadoop-impala-1-4-widens-the-performance-gap/.

[55] K. Ousterhout, R. Rasti, S. Ratnasamy, S. Shenker, and B.-G. Chun. Making sense of performance in data analytics frameworks. In *Proceedings of the 12th USENIX Conference on Networked Systems Design and Implementation*, NSDI '15, 2015.

[56] 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, 2013.

[57] 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 *Proceedings of the 2009 ACM SIGMOD International Conference on Management of Data*, SIGMOD '09, 2009.

[58] Performance comparison - c++/java/python/ruby/jython/jruby/groovy. http://blog.dhananjaynene.com/2008/07/performance-comparison-c-java-python-ruby-jython-jruby-groovy/.

[59] Project tungsten: Bringing spark closer to bare metal. https://databricks.com/blog/2015/04/28/project-tungsten-bringing-spark-closer-to-bare-metal.html.

[60] C. Ranger, R. Raghuraman, A. Penmetsa, G. Bradski, and C. Kozyrakis. Evaluating mapreduce for multi-core and multiprocessor systems. In *Proceedings of the 2007 IEEE 13th International Symposium on High Performance Computer Architecture*, HPCA '07, 2007.

[61] A. Rasmussen, V. T. Lam, M. Conley, G. Porter, R. Kapoor, and A. Vahdat. Themis: An i/o-efficient mapreduce. In *Proceedings of the Third ACM Symposium on Cloud Computing*, SoCC '12, 2012.

[62] B. Saha, H. Shah, S. Seth, G. Vijayaraghavan, A. Murthy, and C. Curino. Apache tez: A unifying framework for modeling and building data processing applications. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, SIGMOD '15, 2015.

[63] J. H. Saltzer. Protection and the control of information sharing in Multics. *Commun. ACM*, 17(7).

[64] Spark. http://spark.apache.org.

[65] Spark will offer interactive querying of live data. https://www.linux.com/news/spark-20-will-offer-interactive-querying-live-data.

[66] Spark SQL performance test. https://github.com/databricks/spark-sql-perf.

[67] Specjbb2015. https://www.spec.org/jbb2015/.

[68] SPECjvm2008. https://www.spec.org/jvm2008/.

[69] StackOverflow: Is Java really slow? http://stackoverflow.com/questions/2163411/is-java-really-slow.

[70] Static initializers will murder your family. http://meowni.ca/posts/static-initializers/.

[71] T. Suganuma, T. Ogasawara, M. Takeuchi, T. Yasue, M. Kawahito, K. Ishizaki, H. Komatsu, and T. Nakatani. Overview of the IBM Java Just-in-time Compiler. *IBM Syst. J.*, 39(1).

[72] T. Suganuma, T. Yasue, M. Kawahito, H. Komatsu, and T. Nakatani. Design and evaluation of dynamic optimizations for a java just-in-time compiler. *ACM Trans. Program. Lang. Syst.*, 27(4).

[73] T. Suganuma, T. Yasue, M. Kawahito, H. Komatsu, and T. Nakatani. A dynamic optimization framework for a java just-in-time compiler. In *Proceedings of the 16th ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications*, OOPSLA '01, 2001.

[74] T. Suganuma, T. Yasue, and T. Nakatani. A region-based compilation technique for a java just-in-time compiler. In *Proceedings of the ACM SIGPLAN 2003 Conference on Programming Language Design and Implementation*, PLDI '03, 2003.

[75] Spark and Tez are successors of MapReduce. http://blogs.gartner.com/nick-heudecker/spark-tez-highlight-mapreduce-problems/.

[76] Tez. https://tez.apache.org/.

[77] A. Thusoo, J. S. Sarma, N. Jain, Z. Shao, P. Chakka, S. Anthony, H. Liu, P. Wyckoff, and R. Murthy. Hive: A warehousing solution over a map-reduce framework. *Proc. VLDB Endow.*, 2(2).

[78] M. Tofte and J.-P. Talpin. Region-based memory management. *Information and Computation*, 132(2):109 – 176, 1997.

[79] Transaction Processing Performance Council (TPC) Benchmark$^{\text{TM}}$DS (TPC-DS): The New Decision Support Benchmark Standard. http://www.tpc.org/tpcds.

[80] L. Wang, J. Zhan, C. Luo, Y. Zhu, Q. Yang, Y. He, W. Gao, Z. Jia, Y. Shi, S. Zhang, C. Zheng, G. Lu, K. Zhan, X. Li, and B. Qiu. Bigdatabench: A big data benchmark suite from internet services. In *Proceedings of the 2014 IEEE 20th International Symposium on High Performance Computer Architecture*, 2014.

[81] Y. Wang, X. Que, W. Yu, D. Goldenberg, and D. Sehgal. Hadoop acceleration through network levitated merge. In *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '11, 2011.

[82] What do real-life apache hadoop workloads look like? `http://blog.cloudera.com/blog/2012/09/what-do-real-life-hadoop-workloads-look-like/`.

[83] Why Java will always be slower than C++. `http://www.jelovic.com/articles/why_java_is_slow.htm`.

[84] 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, 2008.

[85] B. Zorn. The measured cost of conservative garbage collection. *Software – Practice & Experience*, 23(7).

EC-Cache: Load-balanced, Low-latency Cluster Caching with Online Erasure Coding

K. V. Rashmi[1], Mosharaf Chowdhury[2], Jack Kosaian[2], Ion Stoica[1], Kannan Ramchandran[1]
[1]*UC Berkeley* [2]*University of Michigan*

Abstract

Data-intensive clusters and object stores are increasingly relying on in-memory object caching to meet the I/O performance demands. These systems routinely face the challenges of popularity skew, background load imbalance, and server failures, which result in severe load imbalance across servers and degraded I/O performance. Selective replication is a commonly used technique to tackle these challenges, where the number of cached replicas of an object is proportional to its popularity. In this paper, we explore an alternative approach using erasure coding.

EC-Cache is a load-balanced, low latency cluster cache that uses online erasure coding to overcome the limitations of selective replication. EC-Cache employs erasure coding by: (i) splitting and erasure coding individual objects during writes, and (ii) late binding, wherein obtaining any k out of $(k+r)$ splits of an object are sufficient, during reads. As compared to selective replication, EC-Cache improves load balancing by more than $3\times$ and reduces the median and tail read latencies by more than $2\times$, while using the same amount of memory. EC-Cache does so using 10% additional bandwidth and a small increase in the amount of stored metadata. The benefits offered by EC-Cache are further amplified in the presence of background network load imbalance and server failures.

1 Introduction

In recent years, in-memory solutions [12, 25, 56, 87, 89] have gradually replaced disk-based solutions [3, 29, 37] as the primary toolchain for high-performance data analytics. The root cause behind the transition is simple: in-memory I/O is orders of magnitude faster than that involving disks. Since the total amount of memory is significantly smaller than that of disks, the design of in-memory solutions boils down to maximizing the number of requests that can be efficiently served from memory. The primary challenges in this regard are: (i) Judiciously determining which data items to cache and which ones to evict. This issue has been well studied in past work [23, 35, 56, 67]. (ii) Increasing the *effective* memory capacity to be able to cache more data. Sampling [12, 16, 52] and compression [15, 27, 53, 79] are some of the popular approaches employed to increase the effective memory capacity. (iii) Ensuring good I/O performance for the cached data in the presence of skewed popularity, background load imbalance, and failures.

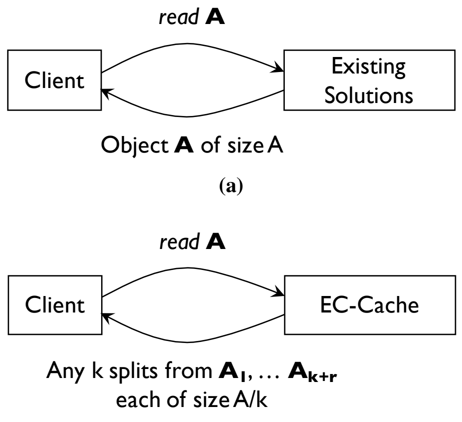


Figure 1: EC-Cache splits individual objects and encodes them using an erasure code to enable read parallelism and late binding during individual reads.

Typically, the popularity of objects in cluster caches are heavily skewed [20, 47], and this creates significant load imbalance across the storage servers in the cluster [20, 48]. The load imbalance necessitates over-provisioning in order to accommodate the peaks in the load distribution, and it also adversely affects the I/O performance. Consequently, load imbalance is one of the key challenges toward improving the performance of cluster caches. In addition to the skew in popularity, massive load fluctuations in the infrastructure due to background activities [33] and failures [70] can result in severe performance degradation.

A popular approach employed to address the aforementioned challenges is selective replication, which replicates objects based on their popularity [20, 63]; that is, it creates more replicas for hot objects. However, due to the limited amount of available memory, selective replication falls short in practice in terms of both load balancing and I/O performance [48].

While typical caches used in web-services and key-value stores cache small-sized objects in the range of a few bytes to few kilobytes, data-intensive cluster caches used for data analytics [23, 56] must store larger objects in the range of tens to hundreds of megabytes (§3). This significant increase in object sizes allows us to take a novel approach, using *erasure coding*, toward load balancing and improving I/O performance in cluster caches.

We present EC-Cache, an in-memory object cache that leverages *online erasure coding* – that is, data is never stored in a decoded form – to provide better load balancing and I/O performance (§4). We show both analytically (§5) and via extensive system evaluation (§6) that EC-Cache can outperform the optimal selective replication mechanism while using the same amount of memory.

EC-Cache employs erasure coding and its properties toward load balancing and improving I/O performance in the following manner.

Self-Coding and Load Spreading: A $(k,\ r)$ erasure code encodes k data units and generates r parity units such that any k of the $(k+r)$ total units are sufficient to decode the original k data units.[1] Erasure coding is traditionally employed in disk-based systems to provide fault-tolerance in a storage-efficient manner. In many such systems [26, 46, 61, 69], erasure coding is applied *across objects*: k objects are encoded to generate r additional objects. Read requests to an object are served from the original object unless it is missing. If the object is missing, it is reconstructed using the parities. In such a configuration, reconstruction using parities incurs huge bandwidth overheads [70]; hence, coding across objects is not useful for load balancing or improving I/O performance. In contrast, EC-Cache divides *individual* objects into k *splits* and creates r additional parity splits. Read requests to an object are served by reading any k of the $(k+r)$ splits and decoding them to recover the desired object (Figure 1). This approach provides multiple benefits. First, spreading the load of read requests across both data and parity splits results in better load balancing under skewed popularity. Second, reading/writing in parallel from multiple splits provides better I/O performance. Third, decoding the object using the parities does not incur any additional bandwidth overhead.

Late Binding: Under self-coding, an object can be reconstructed from *any* k of its $(k+r)$ splits. This allows us to leverage the power of choices: instead of reading exactly k splits, we read $(k+\Delta)$ splits (where $\Delta \leq r$) and wait for the reading of *any* k splits to complete. This late binding makes EC-Cache resilient to background load imbalance and unforeseen stragglers that are common in large clusters [24, 91], and it plays a critical role in taming tail latencies. Note that, while employing object splitting (that is, dividing each object into splits) together with selective replication can provide the benefits of load balancing and opportunities for read parallelism, this approach cannot exploit late binding without incurring high memory and bandwidth overheads (§ 2.3).

We have implemented EC-Cache over Alluxio [56] using Intel's ISA-L library [9]. It can be used as a caching layer on top of object stores such as Amazon S3 [2], Windows Azure Storage [30], and OpenStack Swift [11] where compute and storage are not collocated. It can also be used in front of cluster file systems such as HDFS [29], GFS [42], and Cosmos [31] by considering each block of a distributed file as an individual object.

We evaluated EC-Cache by deploying it on Amazon EC2 using synthetic workloads and production workload traces from a 3000-machine cluster at Facebook. EC-Cache improves the median and tail latencies for reads by more than $2\times$ in comparison to the optimal selective replication scheme; it improves load balancing by more than $3\times$,[2] while using the same amount of memory. EC-Cache's latency reductions increase as objects grow larger: for example, $1.33\times$ for 1 MB objects and $5.5\times$ for 100 MB objects. We note that using $k = 10$ and $\Delta = 1$ suffices to avail these benefits. In other words, a bandwidth overhead of at most 10% can lead to more than 50% reduction in the median and tail latencies. EC-Cache outperforms selective replication by even higher margins in the presence of an imbalance in the background network load and in the presence of server failures. Finally, EC-Cache performs well over a wide range of parameter settings.

Despite its effectiveness, our current implementation of EC-Cache offers advantages only for objects greater than 1 MB due to the overhead of creating $(k+\Delta)$ parallel TCP connections for each read. However, small objects form a negligible fraction of the footprint in many data-intensive workloads (§3). Consequently, EC-Cache simply uses selective replication for objects smaller than this threshold to minimize the overhead. Furthermore, EC-Cache primarily targets immutable objects, which is a popular model in many data analytics systems and object stores. Workloads with frequent, in-place updates are not suitable for EC-Cache because they would require updating all the parity splits of the updated objects.

Finally, we note that erasure codes are gaining increasing popularity in disk-based storage systems for providing fault tolerance in a space-efficient manner [26, 46, 61, 69]. EC-Cache demonstrates the effectiveness of erasure coding for a new setting – in-memory object caching – and toward new goals – improving load balancing and latency characteristics.

[1]Not all erasure codes have this property, but for simplicity, we do not make this distinction.

[2]This evaluation is in terms of the percent imbalance metric described in Section 6.

2 Background and Motivation

This section provides a brief overview of object stores (e.g., Amazon S3 [2], Windows Azure Storage [30], OpenStack Swift [11], and Ceph [86]) and in-memory caching solutions (e.g., Tachyon/Alluxio [56]) used in modern data-intensive clusters. We discuss the tradeoffs and challenges faced therein, followed by the opportunities for improvements over the state-of-the-art.

2.1 Cluster Caching for Object Stores

Cloud object stores [2, 11, 30, 86] provide a simple PUT/GET interface to store and retrieve arbitrary objects at an attractive price point. In recent years, due to the rapid increase in datacenter bandwidth [4, 77], cloud tenants are increasingly relying on these object stores as their primary storage solutions instead of compute-collocated cluster file systems such as HDFS [29]. For example, Netflix has been exclusively using Amazon S3 since 2013 [7]. Separating storage from compute in this manner mitigates disk locality challenges [62]. However, existing object stores can rarely offer end-to-end non-blocking connectivity without storage-side disks becoming a bottleneck. As a result, in-memory storage systems [56] are often used for caching in the compute side.

EC-Cache primarily targets storage-side caching to provide high I/O performance while mitigating the need for compute-side caching. Note that, in the presence of very high-speed networks, it can also be used in environments where compute and storage are collocated.

2.2 Challenges in Object Caching

In-memory object caches face unique tradeoffs and challenges due to workload variations and dynamic infrastructure in large-scale deployments.

Popularity Skew Recent studies from production clusters show that the popularity of objects in cluster caches are heavily skewed [20, 47], which creates significant load imbalance across storage servers in the cluster. This hurts I/O performance and also requires over-provisioning the cluster to accommodate the peaks in the load distribution. Unsurprisingly, load imbalance has been reported to be one of the key challenges toward improving the performance of cluster caches [45, 48].

Background Load Imbalance Across the Infrastructure In addition to skews in object popularity, network interfaces – and I/O subsystems in general – throughout the cluster experience massive load fluctuations due to background activities [33]. Predicting and reacting to these variations in time is difficult. Even with selective replication, performance can deteriorate significantly if the source of an object suddenly becomes a hotspot (§6.3).

Tradeoff Between Memory Efficiency, Fault Tolerance, and I/O Performance In caches, fault tolerance and I/O performance are inherently tied together since failures result in disk I/O activities, which, in turn, significantly increases latency. Given that memory is a constrained and expensive resource, existing solutions either sacrifice fault tolerance (that is, no redundancy) to increase memory efficiency [56, 89], or incur high memory overheads (e.g., replication) to provide fault tolerance [81, 90].

2.3 Potential for Benefits

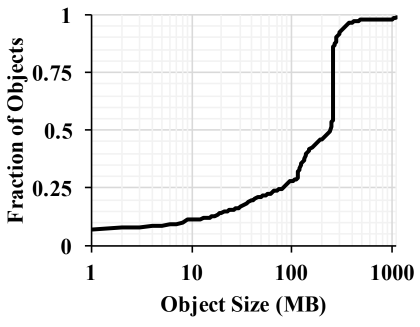
Due to the challenges of popularity skew, background load imbalance, and failures, maintaining a single copy of each object in memory is often insufficient for acheiving high performance. Replication schemes that treat all objects alike do not perform well under popularity skew as they waste memory by replicating less-popular objects. Selective replication [20, 45, 63], where additional replicas of hot objects are cached, only provides coarse-grained support: each replica incurs an additional memory overhead of $1\times$. Selective replication has been shown to fall short in terms of both load balancing and I/O performance [48] (§6.2).

Selective replication along with object splitting (all splits of the same object have the same replication factor) does not solve the problem either. While such an object-splitting approach provides better load balancing and opportunities for read parallelism, it cannot exploit late binding without incurring high memory and bandwidth overheads. As shown in Section 6.6.2, contacting multiple servers to read the splits severely affects tail latencies, and late binding is necessary to rein them in. Hence, under selective replication with object splitting, each object will need at least $2\times$ memory overhead, and, in order to make use of late binding, one must read multiple copies of each of the splits of the object, resulting in at least $2\times$ bandwidth overhead.

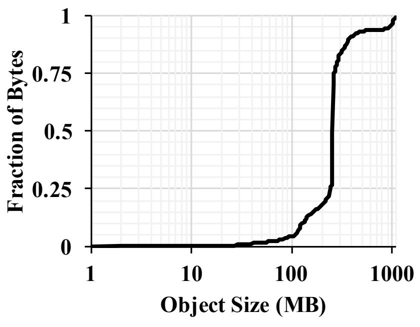
3 Analysis of Production Workload

Object stores are gaining popularity as the primary data storage solution for data analytics pipelines (e.g., at Netflix [6, 7]). As EC-Cache is designed to cater to these use cases, in order to obtain a better understanding of the requirements, we analyzed a trace with millions of reads in a 3000-machine analytics cluster at Facebook. The trace was collected in October 2010, and consists of a mix of batch and interactive MapReduce analytics jobs generated from Hive queries. The block size for the HDFS installation in this cluster was 256 MB, and the corresponding network had a $10:1$ oversubscription ratio.

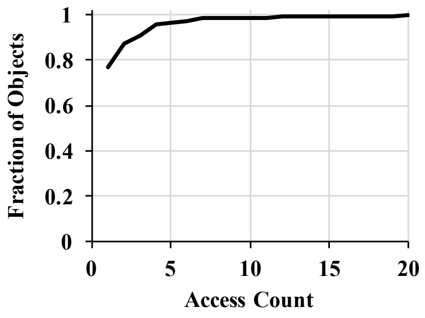
Our goal in analyzing these traces is to highlight characteristics – distributions of object sizes, their relative



(a) Object size



(b) Object footprint



(c) Object access characteristics [23]

Figure 2: Characteristics of object reads in the Facebook data analytics cluster. We observe that (a) large object sizes are more prevalent; (b) small objects have even smaller footprint; and (c) access patterns across objects is heavily skewed. Note that the X-axes are in log-scale in (a) and (b).

impact, access characteristics, and the nature of imbalance in I/O utilizations – that enable us to make realistic assumptions in our analysis, design, and evaluation.

3.1 Large Object Reads are Prevalent

Data-intensive jobs in production clusters are known to follow skewed distributions in terms of their input and output size [23, 32, 34]. We observe a similar skewed pattern in the Facebook trace (Figure 2): only 7% (11%) of the reads are smaller than 1 (10) MB, but their total size in terms of storage usage is miniscule. Furthermore, 28% of the objects are less than 100 MB in size with less than 5% storage footprint. Note that a large fraction of the blocks in the Facebook cluster are 256 MB in size, which corresponds to the vertical segment in Figure 2a.

3.2 Popularity of Objects is Skewed

Next, we focus on object popularity/access patterns. As noted in prior work [20, 23, 32, 56, 61], object popularity follows a Zipf-like skewed pattern; that is, a small fraction of the objects are highly popular. Figure 2c [23, Figure 9] plots the object access characteristics. Note that this measurement does not include objects that were never accessed. Here, the most popular 5% of the objects are seven times more popular than the bottom three-quarters [23].

3.3 Network Load Imbalance is Inherent

As observed in prior studies [28, 33, 44, 51], we found that datacenter traffic across the oversubscribed links can be significantly imbalanced. Furthermore, network imbalances are time varying. The root causes behind such imbalances include, among others, skew in application-level communication patterns [28, 51, 55], rolling upgrades and maintenance operations [28], and imperfect load balancing inside multipath datacenter networks [19]. We measured the network imbalance as the ratio of the maximum and the average utilizations across all

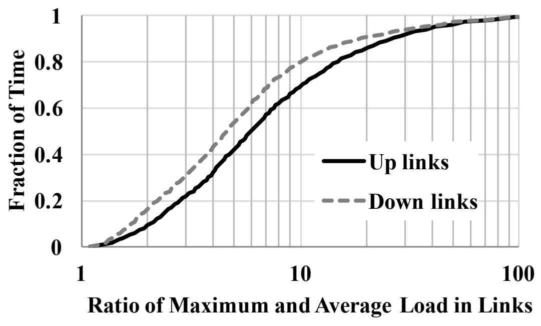


Figure 3: Imbalance in utilizations (averaged over 10-second intervals) of up and down oversubscribed links in Facebook clusters due to data analytics workloads. The X-axis is in log-scale.

oversubscribed links[3] in the Facebook cluster (Figure 3). This ratio was above $4.5\times$ more than 50% of the time for both up and downlinks, indicating significant imbalance. Moreover, the maximum utilization was high for a large fraction of the time, thereby increasing the possibility of congestion. For instance, the maximum uplink utilization was more than 50% of the capacity for more than 50% of the time. Since operations on object stores must go over the network, network hotspots can significantly impact their performance. This impact is amplified for in-memory object caches, where the network is the primary bottleneck.

4 EC-Cache Design Overview

This section provides a high-level overview of EC-Cache's architecture.

4.1 Overall Architecture

EC-Cache is an object caching solution to provide high I/O performance in the presence of popularity skew, background load imbalance, and server failures. It con-

[3]Links connecting top-of-the-rack (ToR) switches to the core.

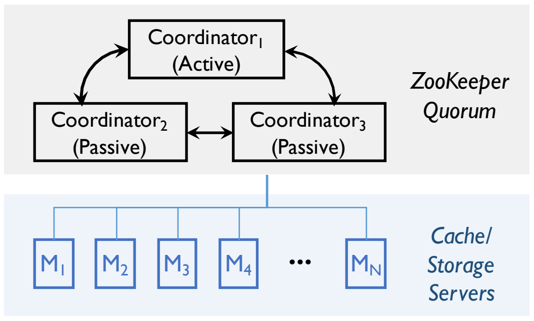


Figure 4: Alluxio architecture

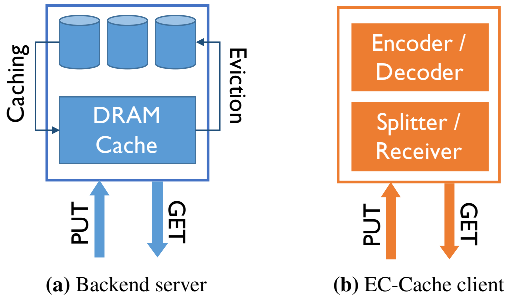


Figure 5: Roles in EC-Cache: (a) backend servers manage interactions between caches and persistent storage for client libraries; (b) EC-Cache clients perform encoding and decoding during writes and reads.

sists of a set of servers, each of which has an in-memory cache on top of on-disk storage. Applications interact with EC-Cache via a client library. Similar to other object stores [2, 11, 30], EC-Cache storage servers are not collocated with the applications using EC-Cache.

We have implemented EC-Cache on top of Alluxio [56], which is a popular caching solution for big data clusters. Consequently, EC-Cache shares some high-level similarities with Alluxio's architecture, such as a centralized architecture with a master coordinating several storage/cache servers (Figure 4).

Backend Storage Servers Both in-memory and on-disk storage in each server is managed by a worker that responds to read and write requests for splits from clients (Figure 5a). Backend servers are unaware of object-level erasure coding introduced by EC-Cache. They also take care of caching and eviction of objects to and from memory using the least-recently-used (LRU) heuristic [56].

EC-Cache Client Library Applications use EC-Cache through a PUT-GET interface (Figure 5b). The client library transparently handles all aspects of erasure coding.

EC-Cache departs significantly from Alluxio in two major ways in its design of the user-facing client library. First, EC-Cache's client library exposes a significantly narrower interface for object-level operations as compared to Alluxio's file-level interface. Second, EC-Cache's client library takes care of splitting and encoding during writes, and reading from splits and decoding during reads instead of writing and reading entire objects.

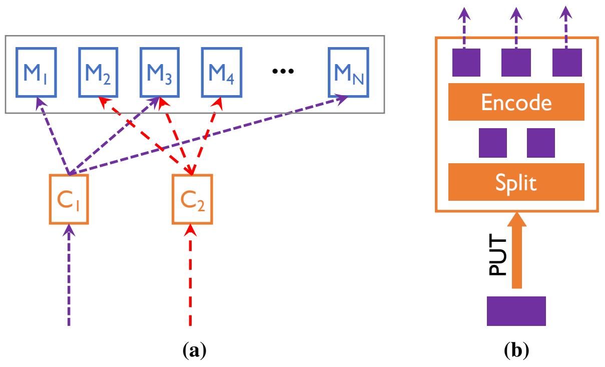


Figure 6: Writes to EC-Cache. (a) Two concurrent writes with $k = 2$ and $r = 1$ from two applications. (b) Steps involved during an individual write inside the EC-Cache client library for an object using $k = 2$ and $r = 1$.

4.2 Writes

EC-Cache stores each object by dividing it into k splits and encoding these splits using a Reed-Solomon code [73] to add r parity splits.[4] It then distributes these $(k+r)$ splits across unique backend servers chosen uniformly at random. Note that each object is allowed to have distinct values of the parameters k and r. Figure 6 depicts an example of object writes with $k = 2$ and $r = 1$ for both objects C_1 and C_2. EC-Cache uses Intel ISA-L [9] for encoding operations.

A key issue in any distributed storage solution is that of data placement and rendezvous, that is, where to write and where to read from. The fact that each object is further divided into $(k+r)$ splits in EC-Cache magnifies this issue. For the same reason, metadata management is also an important issue in our design. Similar to Alluxio and most other storage systems [29, 42, 56], the EC-Cache coordinator determines and manages the locations of all the splits. Each write is preceded by an interaction with the coordinator server that determines where each of the $(k + r)$ splits are to be written. Similarly, each reader receives the locations of the splits through a single interaction with the coordinator.

EC-Cache requires a minimal amount of additional metadata to support object splitting. For each object, EC-Cache stores its associated k and r values and the associated $(k + r)$ server locations (32-bit unsigned integers). This forms only a small fraction of the total metadata size of an object.

[4] Section 4.4 discusses the choice of the erasure coding scheme.

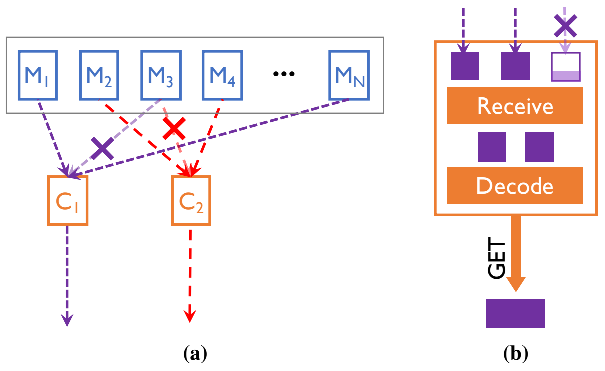


Figure 7: Reads from EC-Cache. (a) Two concurrent reads with $k = 2$ and $r = 1$. Reads from M_3 are slow and hence are ignored (crossed). (b) Steps involved in an individual read in the client library for an object with $k = 2, r = 1$, and $\Delta = 1$.

4.3 Reads

The key advantage of EC-Cache comes into picture during read operations. Instead of reading from a single replica, the EC-Cache client library reads from $(k + \Delta)$ splits in parallel chosen uniformly at random (out of the $(k + r)$ total splits of the object). This provides three benefits. First, it exploits I/O parallelism. Second, it distributes the load across many backend servers helping in balancing the load. Third, the read request can be completed as soon as any k out of $(k + \Delta)$ splits arrive, thereby avoiding stragglers. Once k splits of an object arrives, the decoding operation is performed using Intel ISA-L [9].

Figure 7a provides an example of a read operation over the objects stored in the example presented in Figure 6. In this example, both the objects have $k = 2$ and $r = 1$. Although 2 splits are enough to complete each read request, EC-Cache issues an additional read (that is, $\Delta = 1$). Since both objects had one split in server M$_3$, reading from that server may be slow. However, instead of waiting for that split, EC-Cache proceeds as soon as it receives the other 2 splits (Figure 7b) and decodes them to complete the object read requests.

Additional reads play a critical role in avoiding stragglers, and thus, in reducing tail latencies. However, they also introduce additional load on the system. The bandwidth overhead due to additional reads is precisely $\frac{\Delta}{k}$. In Section 6.6.2, we present a sensitivity analysis with respect to Δ, highlighting the interplay between the above two aspects.

4.4 Choice of Erasure Code

In disk-based storage systems, erasure codes are employed primarily to provide fault tolerance in a storage-efficient manner. In these systems, network and I/O resources consumed during recovery of failed or otherwise unavailable data units play a critical role in the choice of the erasure code employed [46, 69, 70]. There has been a considerable amount of recent work on designing erasure codes for distributed storage systems to optimize recovery operations [26, 43, 68, 71, 72]. Many distributed storage systems are adopting these recovery-optimized erasure codes in order to reduce network and I/O consumption [8, 46, 69]. On the other hand, EC-Cache employs erasure codes for load balancing and improving read performance of cached objects. Furthermore, in this caching application, recovery operations are not a concern as data is persisted in the underlying storage layer.

We have chosen to use Reed-Solomon (RS) [73] codes for two primary reasons. First, RS codes are Maximum-Distance-Separable (MDS) codes [59]; that is, they possess the property that any k out of the $(k + r)$ splits are sufficient to decode the object. This property provides maximum flexibility in the choice of splits for load balancing and late binding. Second, the Intel ISA-L [9] library provides a highly optimized implementation of RS codes that significantly decreases the time taken for encoding and decoding operations. This reduced decoding complexity makes it feasible for EC-Cache to perform decoding for every read operation. Both the above factors enable EC-Cache to exploit properties of erasure coding to achieve significant gains in load balancing and read performance (§6).

5 Analysis

In this section, we provide an analytical explanation for the benefits offered by EC-Cache.

5.1 Impact on Load Balancing

Consider a cluster with S servers and F objects. For simplicity, let us first assume that all objects are equally popular. Under selective replication, each object is placed on a server chosen uniformly at random out of the S servers. For simplicity, first consider that EC-Cache places each split of a object on a server chosen uniformly at random (neglecting the fact that each split is placed on a unique server). The total load on a server equals the sum of the loads on each of the splits stored on that server. Thus the load on each server is a random variable. Without loss of generality, let us consider the load on any particular server and denote the corresponding random variable by L.

The variance of L directly impacts the load imbalance in the cluster – intuitively, a higher variance of L implies a higher load on the maximally loaded server in comparison to the average load; consequently, a higher load imbalance.

Under this simplified setting, the following result holds.

Theorem 1 For the setting described above:

$$\frac{\text{Var}(L_{\text{EC-Cache}})}{\text{Var}(L_{\text{Selective Replication}})} = \frac{1}{k}.$$

Proof: Let $w > 0$ denote the popularity of each of the files. The random variable $L_{\text{Selective Replication}}$ is distributed as a Binomial random variable with F trials and success probability $\frac{1}{S}$, scaled by w. On the other hand, $L_{\text{EC-Cache}}$ is distributed as a Binomial random variable with kF trials and success probability $\frac{1}{S}$, scaled by $\frac{w}{k}$. Thus we have

$$\frac{\text{Var}(L_{\text{EC-Cache}})}{\text{Var}(L_{\text{Selective Replication}})} = \frac{\left(\frac{w}{k}\right)^2 (kF)\frac{1}{S}\left(1-\frac{1}{S}\right)}{w^2 F \frac{1}{S}\left(1-\frac{1}{S}\right)} = \frac{1}{k},$$

thereby proving our claim. □

Intuitively, the splitting action of EC-Cache leads to a smoother load distribution in comparison to selective replication. One can further extend Theorem 1 to accommodate a skew in the popularity of the objects. Such an extension leads to an identical result on the ratio of the variances. Additionally, the fact that each split of an object in EC-Cache is placed on a unique server further helps in evenly distributing the load, leading to even better load balancing.

5.2 Impact on Latency

Next, we focus on how object splitting impacts read latencies. Under selective replication, a read request for an object is served by reading the object from a server. We first consider naive EC-Cache without any additional reads. Under naive EC-Cache, a read request for an object is served by reading k of its splits in parallel from k servers and performing a decoding operation. Let us also assume that the time taken for decoding is negligible compared to the time taken to read the splits.

Intuitively, one may expect that reading splits in parallel from different servers will reduce read latencies due to the parallelism. While this reduction indeed occurs for the average/median latencies, the tail latencies behave in an opposite manner due to the presence of stragglers – one slow split read delays the completion of the entire read request.

In order to obtain a better understanding of the aforementioned phenomenon, let us consider the following simplified model. Consider a parameter $p \in [0, 1]$ and assume that for any request, a server becomes a straggler with probability p, independent of all else. There are two primary contributing factors to the distributions of the latencies under selective replication and EC-Cache:

(a) Proportion of stragglers: Under selective replication, the fraction of requests that hit stragglers is p. On the other hand, under EC-Cache, a read request for an object will face a straggler if any of the k servers from where splits are being read becomes a straggler. Hence, a higher fraction $\left(1-(1-p)^k\right)$ of read requests can hit stragglers under naive EC-Cache.

(b) Latency conditioned on absence/presence of stragglers: If a read request does not face stragglers, the time taken for serving a read request is significantly smaller under EC-Cache as compared to selective replication because splits can be read in parallel. On the other hand, in the presence of a straggler in the two scenarios, the time taken for reading under EC-Cache is about as large as that under selective replication.

Putting the aforementioned two factors together we get that the relatively higher likelihood of a straggler under EC-Cache increases the number of read requests incurring a higher latency. The read requests that do not encounter any straggler incur a lower latency as compared to selective replication. These two factors explain the decrease in the median and mean latencies, and the increase in the tail latencies.

In order to alleviate the impact on tail latencies, we use additional reads and late binding in EC-Cache. Reed-Solomon codes have the property that any k of the collection of all splits of an object suffice to decode the object. We exploit this property by reading more than k splits in parallel, and using the k splits that are read first. It is well known that such additional reads help in mitigating the straggler problem and alleviate the affect on tail latencies [36, 82].

6 Evaluation

We evaluated EC-Cache through a series of experiments on Amazon EC2 [1] clusters using synthetic workloads and traces from Facebook production clusters. The highlights of the evaluation results are:

- For skewed popularity distributions, EC-Cache improves load balancing over selective replication by 3.3× while using the same amount of memory. EC-Cache also decreases the median latency by 2.64× and the 99.9th percentile latency by 1.79× (§6.2).
- For skewed popularity distributions *and* in the presence of background load imbalance, EC-Cache decreases the 99.9th percentile latency w.r.t. selective replication by 2.56× while maintaining the same benefits in median latency and load balancing as in the case without background load imbalance (§6.3).
- For skewed popularity distributions *and* in the presence of server failures, EC-Cache provides a graceful degradation as opposed to the significant degradation in tail latency faced by selective replication. Specifically, EC-Cache decreases the 99.9th percentile latency w.r.t. selective replication by 2.8× (§6.4).
- EC-Cache's improvements over selective replication increase as object sizes increase in production traces;

e.g., 5.5× at median for 100 MB objects with an upward trend (§6.5).

- EC-Cache outperforms selective replication across a wide range of values of k, r, and Δ (§6.6).

6.1 Methodology

Cluster Unless otherwise specified, our experiments use 55 `c4.8xlarge` EC2 instances. 25 of these machines act as the backend servers for EC-Cache, each with 8 GB cache space, and 30 machines generate thousands of read requests to EC-Cache. All machines were in the same Amazon Virtual Private Cloud (VPC) with 10 Gbps enhanced networking enabled; we observed around 4-5 Gbps bandwidth between machines in the VPC using `iperf`.

As mentioned earlier, we implemented EC-Cache on Alluxio [56], which, in turn, used Amazon S3 [2] as its persistence layer and runs on the 25 backend servers. We used DFS-Perf [5] to generate the workload on the 30 client machines.

Metrics Our primary metrics for comparison are *latency* in reading objects and *load imbalance* across the backend servers.

Given a workload, we consider mean, median, and high-percentile latencies. We measure improvements in latency as:

$$\text{Latency Improvement} = \frac{\text{Latency w/ Compared Scheme}}{\text{Latency w/ EC-Cache}}$$

If the value of this "latency improvement" is greater (or smaller) than one, EC-Cache is better (or worse).

We measure load imbalance using the percent imbalance metric λ defined as follows:

$$\lambda = \left(\frac{L_{\max} - L_{\text{avg}^\star}}{L_{\text{avg}^\star}}\right) * 100, \qquad (1)$$

where $L_{\max}$ is the load on the server which is maximally loaded and $L_{\text{avg}^\star}$ is the load on any server under an *oracle* scheme, where the total load is equally distributed among all the servers without any overhead. λ measures the percentage of additional load on the maximally loaded server as compared to the ideal average load. Because EC-Cache operates in the bandwidth-limited regime, the load on a server translates to the total amount of data read from that server. Lower values of λ are better. Note that the percent imbalance metric takes into account the additional load introduced by EC-Cache due to additional reads.

Setup We consider a Zipf distribution for the popularity of objects, which is common in many real-world object popularity distributions [20, 23, 56]. Specifically, we consider the Zipf parameter to be 0.9 (that is, high skew).

Unless otherwise specified, we allow both selective replication and EC-Cache to use 15% memory overhead to handle the skew in the popularity of objects. Selective replication uses all the allowed memory overhead for handling popularity skew. Unless otherwise specified, EC-Cache uses $k = 10$ and $\Delta = 1$. Thus, 10% of the allowed memory overhead is used to provide one parity to each object. The remaining 5% is used for handling popularity skew. Both schemes make use of the skew information to decide how to allocate the allowed memory among different objects in an identical manner: the number of replicas for an object under selective replication and the number of additional parities for an object under EC-Cache are calculated so as to flatten out the popularity skew to the extent possible starting from the most popular object, until the memory budget is exhausted.

Moreover, both schemes use uniform random placement policy to evenly distribute objects (splits in case of EC-Cache) across memory servers.

Unless otherwise specified, the size of each object considered in these experiments is 40 MB. We present results for varying object sizes observed in the Facebook trace in Section 6.5. In Section 6.6, we perform a sensitivity analysis with respect to all the above parameters.

Furthermore, we note that while the evaluations presented here are for the setting of high skew in object popularity, EC-Cache outperforms selective replication in scenarios with low skew in object popularity as well. Under high skew, EC-Cache offers significant benefits in terms of load balancing and read latency. Under low skew, while there is not much to improve in load balancing, EC-Cache will still provide latency benefits.

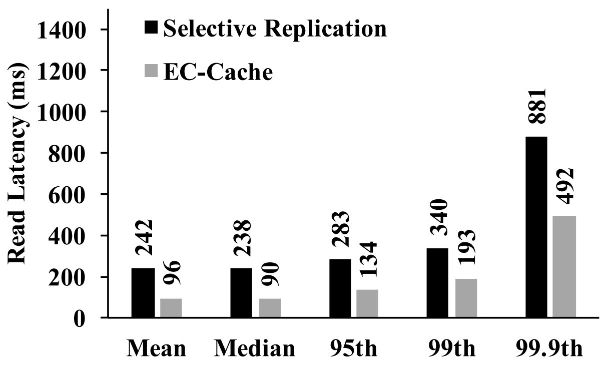
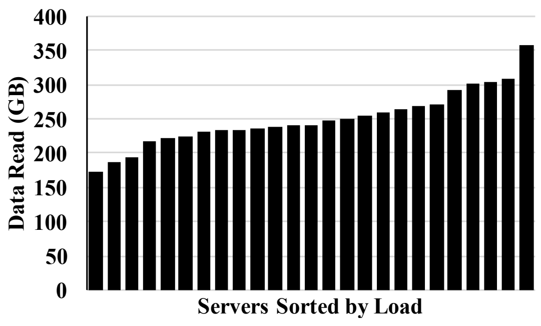


Figure 8: Read latencies under skewed popularity of objects.

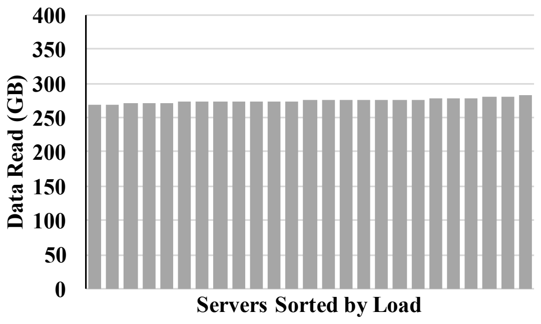
6.2 Skew Resilience

We begin by evaluating the performance of EC-Cache in the presence of skew in object popularity.

Latency Characteristics Figure 8 compares the mean, median, and tail latencies of EC-Cache and selective replication. We observe that EC-Cache improves median and mean latencies by 2.64× and 2.52×, respectively. EC-Cache outperforms selective replication at high per-



(**a**) Selective replication



(**b**) EC-Cache with $k = 10$

Figure 9: Comparison of load distribution across servers in terms of the amount of data read from each server. The percent imbalance metric λ for selective replication and EC-Cache are 43.45% and 13.14% respectively.

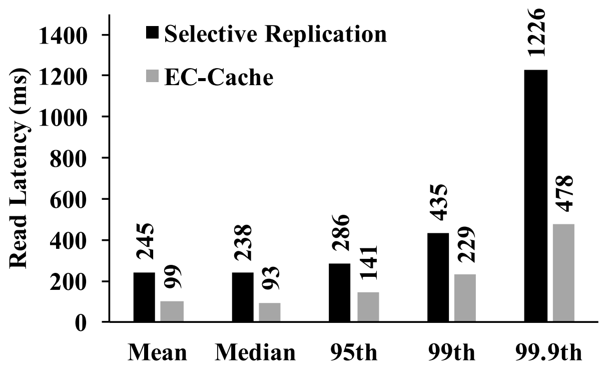


Figure 10: Read latencies in the presence of background traffic from big data workload.

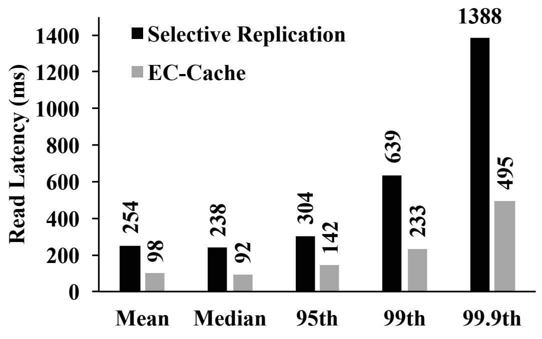


Figure 11: Read latencies in the presence of server failures.

centiles as well, improving the latency by $1.76\times$ at the 99th percentile and by $1.79\times$ at the 99.9th percentile.

Load Balancing Characteristics Figure 9 presents the distribution of loads across servers. The percent imbalance metric λ observed for selective replication and EC-Cache in this experiment are 43.45% and 13.14% respectively.

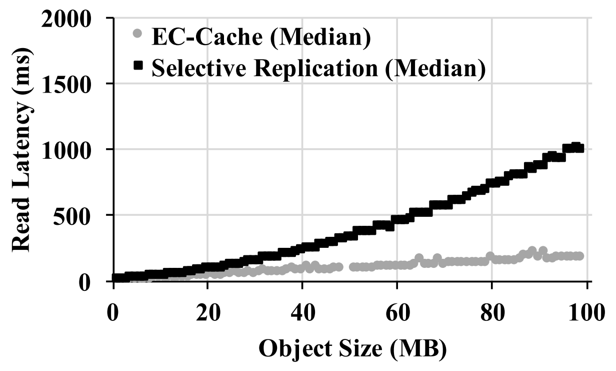
Decoding Overhead During Reads We observed that the time taken to decode during the reads is approximately 30% of the total time taken to complete a read request. Despite this overhead, we see (Figure 8) that EC-Cache provides a significant reduction in both median and tail latencies. Although our current implementation uses only a single thread for decoding, the underlying erasure codes permit the decoding process to be made embarrassingly parallel, potentially allowing for a linear speed up; this, in turn, can further improve EC-Cache's latency characteristics.

6.3 Impact of Background Load Imbalance

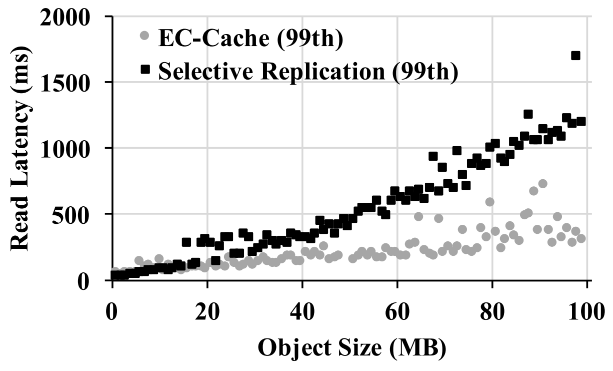
We now investigate EC-Cache's performance in the presence of a background network load, specifically in the presence of unbalanced background traffic. For this experiment, we generated a background load that follows traffic characteristics similar to those described in Section 3.3. Specifically, we emulated network transfers from shuffles for the jobs in the trace. Shuffles arrive following the same arrival pattern of the trace. For each shuffle, we start some senders (emulating mappers) and receivers (emulating reducers) that transfer randomly generated data over the network. The amount of data received by each receiver for each shuffle followed a distribution similar to that in the trace.

Latency Characteristics Figure 10 compares the mean, median, and tail latencies using both EC-Cache and selective replication. We observe that as in Section 6.2, EC-Cache improves the median and mean latencies by $2.56\times$ and $2.47\times$ respectively.

At higher percentiles, EC-Cache's benefits over selective replication are even more than that observed in Section 6.2. In particular, EC-Cache outperforms selective replication by $1.9\times$ at the 99th percentile and by $2.56\times$ at the 99.9th percentile. The reason for these improvements is the following: while selective replication gets stuck in few of the overloaded backend servers, EC-Cache remains almost impervious to such imbalance due to late binding.



(a) Median latency



(b) 99th percentile latency

Figure 12: Comparison of EC-Cache and selective replication read latencies over varying object sizes in the Facebook production trace. EC-Cache's advantages improve as objects become larger.

Load Balancing Characteristics The percent imbalance metric λ for selective replication and EC-Cache are similar to that reported in Section 6.2. This is because the imbalance in background load does not affect the load distribution across servers due to read requests.

6.4 Performance in Presence of Failures

We now evaluate the performance of EC-Cache in the presence of server failures. This experiment is identical to that in Section 6.2 except with one of the back-end servers terminated. The read latencies in this degraded mode are shown in Figure 11. Comparing the latencies in Figure 8 and Figure 11, we see that the performance of EC-Cache does not degrade much as most objects are still served from memory. On the other hand, selective replication suffers significant degradation in tail latencies as some of the objects are now served from the underlying storage system. Here, EC-Cache outperforms selective replication by 2.7× at the 99th percentile and by 2.8× at the 99.9th percentile.

6.5 Performance on Production Workload

So far we focused on EC-Cache's performance for a fixed object size. In this section, we compare EC-Cache against selective replication for varying object sizes based on the workload collected from Facebook (details in Section 3).

Figure 12 presents the median and the 99th percentile read latencies for objects of different sizes (starting from 1 MB). Note that EC-Cache resorts to selective replication for objects smaller than 1 MB to avoid communication overheads.

We make two primary observations. First, EC-Cache's median improvements over selective replication steadily increases with the object size; e.g., EC-Cache is 1.33× faster for 1 MB-sized objects, which improves to 5.5× for 100 MB-sized objects and beyond. Second, EC-Cache's 99th percentile improvements over selective replication kick off when object sizes grow beyond 10 MB. This is because EC-Cache's constant overhead of establishing $(k + \Delta)$ connections is more pronounced for smaller reads, which generally have lower latencies. Beyond 10 MB, connection overheads get amortized due to increased read latency, and EC-Cache's improvements over selective replication even in tail latencies steadily increase from 1.25× to 3.85× for 100 MB objects.

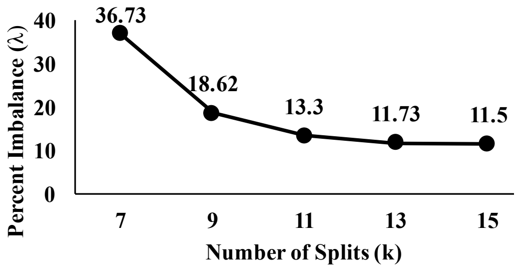


Figure 13: Load imbalance for varying values of k with $\Delta = 1$: percent imbalance metric (λ) decreases as objects are divided into more splits.

6.6 Sensitivity Evaluation

In this section, we evaluate the effects of the choice of different EC-Cache parameters. We present the results for 10 MB objects (instead of 40 MB as in prior evaluations) in order to bring out the effects of all the parameters more clearly and to be able to sweep for a wide range of parameters.

6.6.1 Number of splits k

Load Balancing Characteristics The percent imbalance metric for varying values of k with $\Delta = 1$ are shown in Figure 13. We observe that load balancing improves with increasing k. There are two reasons for this phenomenon: (i) A higher value of k leads to a smaller granularity of individual splits, thereby resulting in a

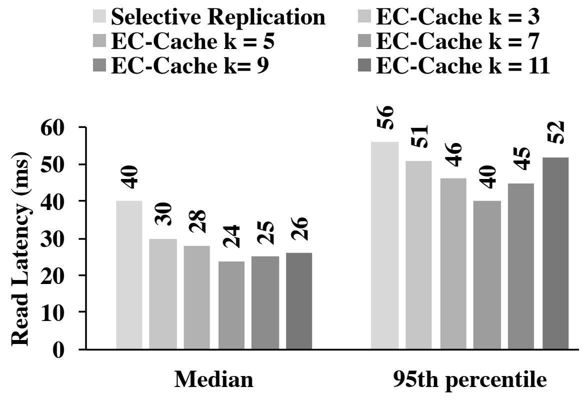


Figure 14: Impact of the number of splits k on the read latency.

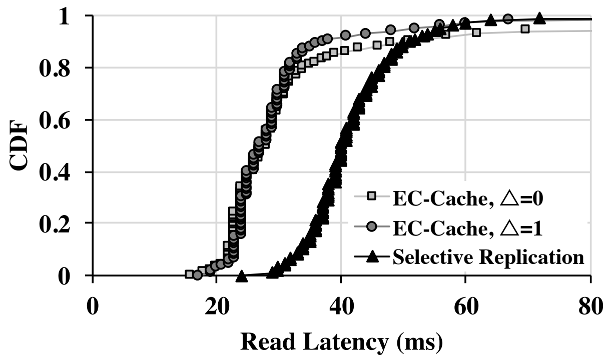


Figure 15: CDF of read latencies showing the need for additional reads in reining in tail latencies in EC-Cache.

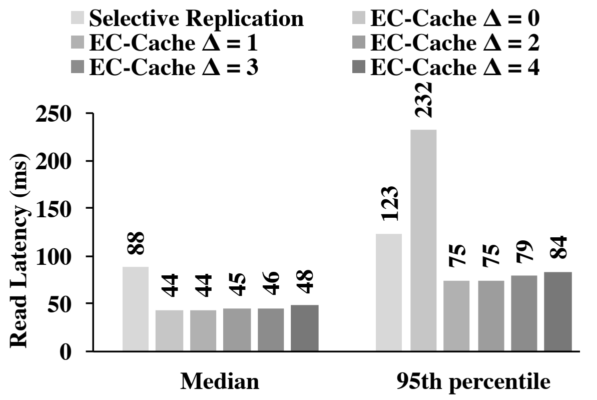


Figure 16: Impact of the number of additional reads on read latency.

greater smoothing of the load under skewed popularity. (ii) With a fixed value of Δ, the load overhead due to additional reads varies inversely with the value of k. This trend conforms to the theoretical analysis presented in Section 5.1.

Latency Characteristics Figure 14 shows a comparison of median and 95th percentile read latencies for varying values of k with $\Delta = 1$. The corresponding values for selective replication are also provided for comparison. We observe that parallelism helps in improving median latencies, but with diminishing returns. However, higher values of k lead to worse tail latencies as a result of the straggler effect discussed earlier in Section 5. Hence, for $k > 10$, more than one additional reads are needed to rein in the tail latencies. We elaborate this effect below.

6.6.2 Additional Reads (Δ)

First, we study the necessity of additional reads. Figure 15 shows the CDF of read latencies from about $160,000$ reads for selective replication and EC-Cache with $k = 10$ with and without additional reads, that is, with $\Delta = 1$ and $\Delta = 0$, respectively. We observe that, without any additional reads, EC-Cache performs quite well in terms of the median latency, but severely suffers at high percentiles. This is due to the effect of stragglers as discussed in Section 5.2. Moreover, adding just one additional read helps EC-Cache tame these negative effects. Figure 15 also shows that selective replication with object splitting (as discussed in Section 2.3) would not perform well.

Next, we study the effect of varying values of Δ. In this experiment, we vary Δ from 0 to 4, set $k = 12$, and use an object size of 20 MB. We choose $k = 12$ instead of 10 because the effect of additional reads is more pronounced for higher values of k, and we choose a larger object size (20 MB instead of 10 MB) because the value of k is higher (§7.4). We use uniform popularity distribution across objects so that each object is provided with equal (specifically, $r = 4$) number of parities. This allows us to evaluate with values of Δ up to 4. Figure 16 shows the impact of different number of additional reads on the read latency. We see that the first one or two additional reads provide a significant reduction in the tail latencies while subsequent additional reads provide little additional benefits. In general, having too many additional reads would start hurting the performance because they would cause a proportional increase in communication and bandwidth overheads.

6.6.3 Memory Overhead

Up until now, we have compared EC-Cache and selective replication with a fixed memory overhead of 15%. Given a fixed amount of total memory, increasing memory overhead allows a scheme to cache more redundant objects but fewer unique objects. In this section, we vary memory overhead and evaluate the latency and load balancing characterisitics of selective replication and EC-Cache.

We observed that the relative difference in terms of latency between EC-Cache and selective replication remained similar to that shown in Figure 8 – EC-Cache provided a significant reduction in the median and tail latencies as compared to selective replication even for higher memory overheads. However, in terms of load

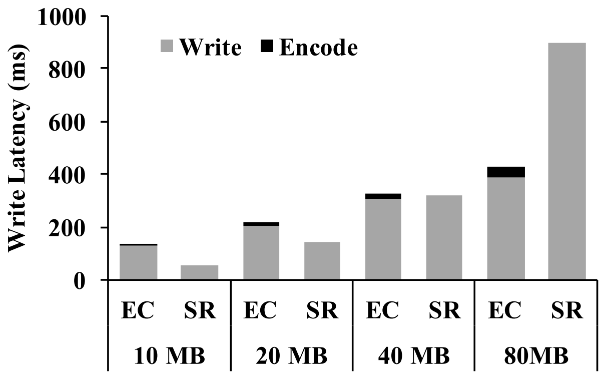


Figure 17: Comparison of writing times (and encoding time for EC-Cache) for different object sizes.

balancing, the gap between EC-Cache and selective replication decreased with increasing memory overhead. This is because EC-Cache was almost balanced even with just 15% memory overhead (Figure 9) with little room for further improvement. In contrast, selective replication became more balanced due to the higher memory overhead allowed, reducing the relative gap from EC-Cache.

6.7 Write Performance

Figure 17 shows a comparison of the average write times. The time taken to write an object in EC-Cache involves the time to encode and the time to write out the splits to different workers; Figure 17 depicts the breakdown of the write time in terms of these two components. We observe that EC-Cache is faster than selective replication when writing objects larger than 40 MB, supplementing its faster performance in terms of the read times observed earlier. EC-Cache performs worse for smaller objects due to the overhead of connecting to several machines in parallel. Finally, we observe that the time taken for encoding is less than 10% of the total write time, regardless of the object size.

7 Discussion

While EC-Cache outperforms existing solutions both in terms of latency and load balancing, our current implementation has several known limitations. We believe that addressing these limitations will further improve EC-Cache's performance.

7.1 Networking Overheads

A key reason behind EC-Cache being less effective for smaller objects is its communication overhead. More specifically, creating many TCP connections accounts for a constant, non-negligible portion (few milliseconds) of a read's duration. This factor is more pronounced for smaller read requests which generally have shorter durations. Using long-running, reusable connections may allow us to support even smaller objects. Furthermore, multiplexing will also help in decreasing the total number of TCP connections in the cluster.

7.2 Reducing Bandwidth Overhead

EC-Cache has 10% bandwidth overhead in our present setup. While this overhead does not significantly impact performance during non-peak hours, it can have a non-negligible impact during the peak. In order to address this issue, one may additionally employ proactive cancellation [36, 64] that can help reduce bandwidth overheads of speculative reads.

7.3 Time Varying Skew

EC-Cache can handle time-varying popularity skew and load imbalance by changing the number of parity splits of objects. However, we have not yet implemented this feature due to a limitation posed by Alluxio. In our current implementation, we store individual splits of an object as part of the file abstraction in Alluxio to reduce metadata overheads (§4.2). Since Alluxio does not currently offer support for appending to a file once the file is closed (`ALLUXIO-25` [14]), we cannot dynamically change the number of parities and adapt to time-varying skew. Assuming the presence of underlying support for appending, we expect EC-Cache to respond to time-varying skews better than selective replication. This is because the overhead of any object can be changed in fractional increments in EC-Cache as opposed to the limitation of having only integral increments in selective replication.

7.4 Choice of parameters

Although EC-Cache performs well for a wide range of parameters in our evaluation (§6.6), we outline a few rules of thumb for choosing its parameter values below.

The value of parameter k is chosen based on the size of the object and cluster characteristics: a higher value of k provides better load balancing but negatively affects tail latencies for too large values (as shown in Figure 13 and Figure 14). In general, the larger the size of an object, the higher the value of k it can accommodate without resulting in too small-sized splits and without adversely affecting the tail latency. In our evaluations, we observed $k = 10$ to perform well for a wide range of object sizes (Figure 12).

Suitable choices for Δ depend on the choice of k. As discussed in Section 6.6.2, a higher value of Δ is needed for higher values of k in order to rein in tail latencies. At the same time, each additional read results in a proportional increase in the bandwidth overhead, which would degrade performance for too large a value. In our evaluations, we observed $\Delta = 1$ to be sufficient for $k = 10$ (Figure 10 and Figure 12).

The value of parameter r for each object is chosen based on the skew in object popularity (§6.1).

8 Related Work

A key focus of this work is to demonstrate and validate a new application of erasure coding, specifically in in-memory caching systems, to achieve load balancing and to reduce the median and tail read latencies. The basic building blocks employed in EC-Cache are simple and have been studied and employed in various other systems and settings. We borrow and build on the large body of existing work in this area. However, to the best of our knowledge, EC-Cache is the first object caching system that employs erasure coding to achieve load balancing and to reduce read latencies.

Caching in Data-Intensive Clusters Given that reading data from disks is often the primary bottleneck in data analytics [3, 24, 37, 49, 56, 88, 89, 91], caching frequently used data has received significant attention in recent years [21, 23, 56, 89]. However, existing caching solutions typically keep a single copy of data to increase the memory capacity, which leaves them vulnerable to popularity skew, background load imbalance, and failures, all of which result in disk accesses.

(Selective) Replication Replication is the most common technique for guarding against performance degradation in the face of popularity skew, background load imbalance, and failures [29, 31, 42, 81]. Giving every object an identical replication factor, however, wastes capacity in the presence of skew, and selective replication [20, 63] forms a better option in this case. However, selective replication has a number of drawbacks (§2.3) that EC-Cache overcomes.

Erasure Coding in Storage Systems For decades, disk arrays have employed erasure codes to achieve space-efficient fault tolerance in RAID systems [65]. The benefits of erasure coding over replication to provide fault tolerance in distributed storage systems has also been well studied [85, 93], and erasure codes have been employed in many related settings such as network-attached-storage systems [18], peer-to-peer storage systems [54, 74], etc. Recently, erasure coding has been widely used for storing relatively *cold* data in datacenter-scale distributed storage systems [46, 61, 86] to achieve fault tolerance while minimizing storage requirements. While some of these storage systems [61, 70, 79] encode across objects, others employ self-coding [80, 86]. However, the purpose of erasure coding in these systems is to achieve storage-efficient fault tolerance, while the focus of EC-Cache is on load balancing and reducing the median and tail read latencies. Aggarwal et al. [17] proposed augmenting erasure-coded disk-based storage systems with a cache at the proxy or client side to reduce latency. In contrast, EC-Cache directly applies erasure coding on objects stored in cluster caches to achieve load balancing and to reduce latency when serving objects from memory.

Late binding Many systems have employed the technique of sending additional/redundant requests or running redundant jobs to rein in tail latency in various settings [22, 36, 40, 66, 78, 83]. The effectiveness of late binding for load balancing and scheduling has been well known and well utilized in many systems [60, 64, 82]. Recently, there have also been a body of theoretical work that analyzes the performance of redundant requests [41, 50, 57, 75, 76, 84].

In-Memory Key-Value Stores A large body of work in recent years has focused on building high-performance in-memory key-value (KV) stores [10, 13, 15, 38, 39, 53, 58, 63]. EC-Cache focuses on a different workload where object sizes are much larger than typical values in these KV stores. However, EC-Cache may be used as a caching layer for holding slabs, where each slab contain many key-value pairs. While KV stores have typically employed replication for fault tolerance, a recent work [92] uses erasure coding to build a fault-tolerant in-memory KV store. The role of erasure coding in [92] is to provide space-efficient fault tolerance, whereas EC-Cache employs erasure coding toward load balancing and reducing the median and tail read latencies.

9 Conclusion

Caching solutions used in conjunction with modern object stores and cluster file systems typically rely on uniform or selective replication that do not perform well in the presence of skew in data popularity, imbalance in network load, or failures of machines and software, all of which are common in large clusters. In EC-Cache, we employ erasure coding to overcome the limitations of selective replication and provide significantly better load balancing and I/O performance for workloads with immutable data.

EC-Cache employs self-coding, where each object is divided into k splits and stored in a $(k+r)$ erasure-coded form. The encoding is such that *any* k of the $(k+r)$ splits are sufficient to read an object. Consequently, EC-Cache can leverage the power of choices through late binding: instead of reading from k splits, it reads from $(k+\Delta)$ splits and completes reading an object as soon as the first k splits arrive. The value of Δ can be as low as 1.

The combination of self-coding and late binding, along with fast encoding/decoding using Intel's ISA-L library, allows EC-Cache to significantly outperform the optimal selective replication solution. For instance, for objects of size 40 MB, EC-Cache outperforms selective replication by $3.3\times$ in terms of cache load balancing, and decreases the median and tail read latencies by more than $2\times$. EC-Cache achieves these improvements while using the same amount of memory as selective replication. The relative performance of EC-Cache improves even more in the presence of background/network load imbalance and server failures, and for larger objects.

In conclusion, while erasure codes are commonly used in disk-based storage systems to achieve fault tolerance in a space-efficient manner, EC-Cache demonstrates their effectiveness in a new setting (in-memory object caching) and toward new goals (load balancing and improving the median and tail read latencies).

10 Acknowledgments

We thank our shepherd, Andrea Arpaci-Dusseau, and the anonymous reviewers for their valuable feedback. This research is supported in part by DHS Award HSHQDC-16-3-00083, NSF CISE Expeditions Award CCF-1139158, DOE Award SN10040 DE-SC0012463, and DARPA XData Award FA8750-12-2-0331, NSF grant 1409135 and gifts from Amazon Web Services,Google, IBM, SAP, The Thomas and Stacey Siebel Foundation, Apple Inc., Arimo, Blue Goji, Bosch, Cisco, Cray, Cloudera, Ericsson, Facebook, Fujitsu, HP, Huawei, Intel, Microsoft, Mitre, Pivotal, Samsung, Schlumberger, Splunk, State Farm and VMware. Mosharaf and Jack were supported in part by National Science Foundation (grants CNS-1563095 and CCF-1629397) and Google.

References

[1] Amazon EC2. http://aws.amazon.com/ec2.

[2] Amazon Simple Storage Service. http://aws.amazon.com/s3.

[3] Apache Hadoop. http://hadoop.apache.org.

[4] AWS Innovation at Scale. https://www.youtube.com/watch?v=JIQETrFC_SQ.

[5] DFS-Perf. http://pasa-bigdata.nju.edu.cn/dfs-perf.

[6] Evolution of the Netflix Data Pipeline. http://techblog.netflix.com/2016/02/evolution-of-netflix-data-pipeline.html.

[7] Hadoop platform as a service in the cloud. http://goo.gl/11zFs.

[8] Implement the Hitchhiker erasure coding algorithm for Hadoop. https://issues.apache.org/jira/browse/HADOOP-11828.

[9] Intel Storage Acceleration Library (Open Source Version). https://goo.gl/zkVl4N.

[10] MemCached. http://www.memcached.org.

[11] OpenStack Swift. http://swift.openstack.org.

[12] Presto: Distributed SQL Query Engine for Big Data. https://prestodb.io.

[13] Redis. http://www.redis.io.

[14] Support append operation after completing a file. https://alluxio.atlassian.net/browse/ALLUXIO-25.

[15] R. Agarwal, A. Khandelwal, and I. Stoica. Succinct: Enabling queries on compressed data. In *NSDI*, 2015.

[16] S. Agarwal, B. Mozafari, A. Panda, H. Milner, S. Madden, and I. Stoica. BlinkDB: Queries with bounded errors and bounded response times on very large data. In *EuroSys*, 2013.

[17] V. Aggarwal, Y.-F. R. Chen, T. Lan, and Y. Xiang. Sprout: A functional caching approach to minimize service latency in erasure-coded storage. In *ICDCS*, 2016.

[18] M. Aguilera, R. Janakiraman, and L. Xu. Using erasure codes efficiently for storage in a distributed system. In *DSN*, 2005.

[19] M. Alizadeh, T. Edsall, S. Dharmapurikar, R. Vaidyanathan, K. Chu, A. Fingerhut, F. Matus, R. Pan, N. Yadav, and G. Varghese. CONGA: Distributed congestion-aware load balancing for datacenters. In *SIGCOMM*, 2014.

[20] G. Ananthanarayanan, S. Agarwal, S. Kandula, A. Greenberg, I. Stoica, D. Harlan, and E. Harris. Scarlett: Coping with skewed popularity content in mapreduce clusters. In *EuroSys*, 2011.

[21] G. Ananthanarayanan, A. Ghodsi, S. Shenker, and I. Stoica. Disk-locality in datacenter computing considered irrelevant. In *HotOS*, 2011.

[22] G. Ananthanarayanan, A. Ghodsi, S. Shenker, and I. Stoica. Why let resources idle? Aggressive cloning of jobs with dolly. In *USENIX HotCloud*, June 2012.

[23] G. Ananthanarayanan, A. Ghodsi, A. Wang, D. Borthakur, S. Kandula, S. Shenker, and I. Stoica. PACMan: Coordinated memory caching for parallel jobs. In *NSDI*, 2012.

[24] G. Ananthanarayanan, S. Kandula, A. Greenberg, I. Stoica, Y. Lu, B. Saha, and E. Harris. Reining in the outliers in mapreduce clusters using Mantri. In *OSDI*, 2010.

[25] M. Armbrust, R. S. Xin, C. Lian, Y. Huai, D. Liu, J. K. Bradley, X. Meng, T. Kaftan, M. J. Franklin, A. Ghodsi, and M. Zaharia. Spark SQL: Relational data processing in Spark. In *SIGMOD*, 2015.

[26] M. Asteris, D. Papailiopoulous, A. Dimakis, R. Vadali, S. Chen, and D. Borthakur. XORing elephants: Novel erasure codes for big data. In *PVLDB*, 2013.

[27] L. A. Barroso, J. Clidaras, and U. Hölzle. The datacenter as a computer: An introduction to the design of warehouse-scale machines. *Synthesis lectures on computer architecture*, 8(3):1–154, 2013.

[28] P. Bodik, I. Menache, M. Chowdhury, P. Mani, D. Maltz, and I. Stoica. Surviving failures in bandwidth-constrained datacenters. In *SIGCOMM*, 2012.

[29] D. Borthakur. The Hadoop distributed file system: Architecture and design. Hadoop Project Website, 2007.

[30] 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 *SOSP*, 2011.

[31] R. Chaiken, B. Jenkins, P. Larson, B. Ramsey, D. Shakib, S. Weaver, and J. Zhou. SCOPE: Easy and efficient parallel processing of massive datasets. In *VLDB*, 2008.

[32] Y. Chen, S. Alspaugh, and R. Katz. Interactive analytical processing in big data systems: A cross-industry study of mapreduce workloads. In *VLDB*, 2012.

[33] M. Chowdhury, S. Kandula, and I. Stoica. Leveraging endpoint flexibility in data-intensive clusters. In *SIGCOMM*, 2013.

[34] M. Chowdhury, Y. Zhong, and I. Stoica. Efficient coflow scheduling with Varys. In *SIGCOMM*, 2014.

[35] A. Cidon, A. Eisenman, M. Alizadeh, and S. Katti. Cliffhanger: Scaling performance cliffs in web memory caches. In *NSDI*, 2016.

[36] J. Dean and L. A. Barroso. The tail at scale. *Communications of the ACM*, 56(2):74–80, 2013.

[37] J. Dean and S. Ghemawat. MapReduce: Simplified data processing on large clusters. In *OSDI*, 2004.

[38] A. Dragojević, D. Narayanan, O. Hodson, and M. Castro. FaRM: Fast remote memory. In *NSDI*, 2014.

[39] B. Fan, D. G. Andersen, and M. Kaminsky. MemC3: Compact and concurrent memcache with dumber caching and smarter hashing. In *NSDI*, 2013.

[40] 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.

[41] K. Gardner, S. Zbarsky, S. Doroudi, M. Harchol-Balter, and E. Hyytia. Reducing latency via redundant requests: Exact analysis. *ACM SIGMETRICS Performance Evaluation Review*, 43(1):347–360, 2015.

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

[43] P. Gopalan, C. Huang, H. Simitci, and S. Yekhanin. On the locality of codeword symbols. *IEEE Transactions on Information Theory*, Nov. 2012.

[44] 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.

[45] Y.-J. Hong and M. Thottethodi. Understanding and mitigating the impact of load imbalance in the memory caching tier. In *SoCC*, 2013.

[46] C. Huang, H. Simitci, Y. Xu, A. Ogus, B. Calder, P. Gopalan, J. Li, and S. Yekhanin. Erasure coding in Windows Azure Storage. In *USENIX ATC*, 2012.

[47] Q. Huang, K. Birman, R. van Renesse, W. Lloyd, S. Kumar, and H. C. Li. An analysis of Facebook photo caching. In *SOSP*, 2013.

[48] Q. Huang, H. Gudmundsdottir, Y. Vigfusson, D. A. Freedman, K. Birman, and R. van Renesse. Characterizing load imbalance in real-world networked caches. In *ACM HotNets*, 2014.

[49] M. Isard, V. Prabhakaran, J. Currey, U. Wieder, K. Talwar, and A. Goldberg. Quincy: Fair scheduling for distributed computing clusters. In *SOSP*, 2009.

[50] G. Joshi, Y. Liu, and E. Soljanin. On the delay-storage trade-off in content download from

coded distributed storage systems. *IEEE JSAC*, 32(5):989–997, 2014.

[51] S. Kandula, S. Sengupta, A. Greenberg, P. Patel, and R. Chaiken. The nature of datacenter traffic: Measurements and analysis. In *IMC*, 2009.

[52] S. Kandula, A. Shanbhag, A. Vitorovic, M. Olma, R. Grandl, S. Chaudhuri, and B. Ding. Quickr: Lazily approximating complex adhoc queries in bigdata clusters. In *SIGMOD*, 2016.

[53] A. Khandelwal, R. Agarwal, and I. Stoica. Blowfish: Dynamic storage-performance tradeoff in data stores. In *NSDI*, 2016.

[54] J. Kubiatowicz, D. Bindel, Y. Chen, S. Czerwinski, P. Eaton, D. Geels, R. Gummadi, S. Rhea, H. Weatherspoon, W. Weimer, et al. Oceanstore: An architecture for global-scale persistent storage. *ACM Sigplan Notices*, 35(11):190–201, 2000.

[55] J. Lee, Y. Turner, M. Lee, L. Popa, S. Banerjee, J.-M. Kang, and P. Sharma. Application-driven bandwidth guarantees in datacenters. In *SIGCOMM*, 2014.

[56] H. Li, A. Ghodsi, M. Zaharia, S. Shenker, and I. Stoica. Tachyon: Reliable, memory speed storage for cluster computing frameworks. In *SoCC*, 2014.

[57] G. Liang and U. Kozat. Fast cloud: Pushing the envelope on delay performance of cloud storage with coding. *arXiv:1301.1294*, Jan. 2013.

[58] H. Lim, D. Han, D. G. Andersen, and M. Kaminsky. MICA: A holistic approach to fast in-memory key-value storage. In *NSDI*, 2014.

[59] S. Lin and D. Costello. *Error control coding*. Prentice-hall Englewood Cliffs, 2004.

[60] M. Mitzenmacher, A. W. Richa, and R. Sitaraman. The power of two random choices: A survey of techniques and results. *Handbook of Randomized Computing*, pages 255–312, 2001.

[61] S. Muralidhar, W. Lloyd, S. Roy, C. Hill, E. Lin, W. Liu, S. Pan, S. Shankar, V. Sivakumar, L. Tang, and S. Kuamr. f4: Facebook's warm BLOB storage system. In *OSDI*, 2014.

[62] E. Nightingale, J. Elson, O. Hofmann, Y. Suzue, J. Fan, and J. Howell. Flat Datacenter Storage. In *OSDI*, 2012.

[63] 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.

[64] K. Ousterhout, P. Wendell, M. Zaharia, and I. Stoica. Sparrow: Distributed, low latency scheduling. In *SOSP*, 2013.

[65] D. A. Patterson, G. Gibson, and R. H. Katz. A case for redundant arrays of inexpensive disks (RAID). In *SIGMOD*, 1988.

[66] M. J. Pitkänen and J. Ott. Redundancy and distributed caching in mobile DTNs. In *MobiArch*, 2007.

[67] Q. Pu, H. Li, M. Zaharia, A. Ghodsi, and I. Stoica. Fairride: Near-optimal, fair cache sharing. In *NSDI*, 2016.

[68] K. Rashmi, P. Nakkiran, J. Wang, N. B. Shah, and K. Ramchandran. Having your cake and eating it too: Jointly optimal erasure codes for I/O, storage, and network-bandwidth. In *FAST 15*, 2015.

[69] K. Rashmi, N. B. Shah, D. Gu, H. Kuang, D. Borthakur, and K. Ramchandran. A hitchhiker's guide to fast and efficient data reconstruction in erasure-coded data centers. In *SIGCOMM*, 2015.

[70] K. V. Rashmi, N. B. Shah, D. Gu, H. Kuang, D. Borthakur, and K. Ramchandran. A solution to the network challenges of data recovery in erasure-coded distributed storage systems: A study on the Facebook warehouse cluster. In *USENIX HotStorage*, 2013.

[71] K. V. Rashmi, N. B. Shah, and P. V. Kumar. Optimal exact-regenerating codes for the MSR and MBR points via a product-matrix construction. *IEEE Transactions on Information Theory*, 57(8):5227–5239, Aug. 2011.

[72] K. V. Rashmi, N. B. Shah, and K. Ramchandran. A piggybacking design framework for read-and download-efficient distributed storage codes. In *IEEE International Symposium on Information Theory*, 2013.

[73] I. Reed and G. Solomon. Polynomial codes over certain finite fields. *Journal of the Society for Industrial and Applied Mathematics*, 8(2):300–304, 1960.

[74] S. Rhea, P. Eaton, D. Geels, H. Weatherspoon, B. Zhao, and J. Kubiatowicz. Pond: The OceanStore prototype. In *FAST*, 2003.

[75] N. B. Shah, K. Lee, and K. Ramchandran. The MDS queue: Analysing the latency performance of erasure codes. In *IEEE International Symposium on Information Theory*, 2014.

[76] N. B. Shah, K. Lee, and K. Ramchandran. When do redundant requests reduce latency? *IEEE Transactions on Communications*, 64(2):715–722, 2016.

[77] A. Singh, J. Ong, A. Agarwal, G. Anderson, A. Armistead, R. Bannon, S. Boving, G. Desai, B. Felderman, P. Germano, A. Kanagala, J. Provost, J. Simmons, E. Tanda, J. Wanderer, U. Hölzle, S. Stuart, and A. Vahdat. Jupiter rising: A decade of Clos topologies and centralized control in Google's datacenter network. SIGCOMM, 2015.

[78] C. Stewart, A. Chakrabarti, and R. Griffith. Zoolander: Efficiently meeting very strict, low-latency SLOs. In *USENIX ICAC*, 2013.

[79] A. Thusoo, Z. Shao, S. Anthony, D. Borthakur, N. Jain, J. S. Sarma, R. Murthy, and H. Liu. Data warehousing and analytics infrastructure at Facebook. In *SIGMOD*, 2010.

[80] C. Ungureanu, B. Atkin, A. Aranya, S. Gokhale, S. Rago, G. Calkowski, C. Dubnicki, and A. Bohra. HydraFS: A high-throughput file system for the HYDRAstor content-addressable storage system. In *FAST*, 2010.

[81] R. van Renesse and F. B. Schneider. Chain replication for supporting high throughput and availability. In *OSDI*, 2004.

[82] S. Venkataraman, A. Panda, G. Ananthanarayanan, M. J. Franklin, and I. Stoica. The power of choice in data-aware cluster scheduling. In *OSDI*, 2014.

[83] A. Vulimiri, O. Michel, P. Godfrey, and S. Shenker. More is less: Reducing latency via redundancy. In *ACM HotNets*, 2012.

[84] D. Wang, G. Joshi, and G. Wornell. Efficient task replication for fast response times in parallel computation. In *SIGMETRICS*, 2014.

[85] H. Weatherspoon and J. D. Kubiatowicz. Erasure coding vs. replication: A quantitative comparison. In *IPTPS*, 2002.

[86] S. A. Weil, S. A. Brandt, E. L. Miller, D. D. Long, and C. Maltzahn. Ceph: A scalable, high-performance distributed file system. In *OSDI*, 2006.

[87] Y. Yu, M. Isard, D. Fetterly, M. Budiu, Ú. Erlingsson, P. K. Gunda, and J. Currey. DryadLINQ: A system for general-purpose distributed data-parallel computing using a high-level language. In *OSDI*, 2008.

[88] M. Zaharia, D. Borthakur, J. Sen Sarma, K. Elmeleegy, S. Shenker, and I. Stoica. Delay scheduling: A simple technique for achieving locality and fairness in cluster scheduling. In *EuroSys*, 2010.

[89] M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauley, M. Franklin, S. Shenker, and I. Stoica. Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. In *NSDI*, 2012.

[90] M. Zaharia, T. Das, H. Li, S. Shenker, and I. Stoica. Discretized streams: Fault-tolerant stream computation at scale. In *SOSP*, 2013.

[91] M. Zaharia, A. Konwinski, A. D. Joseph, R. Katz, and I. Stoica. Improving MapReduce performance in heterogeneous environments. In *OSDI*, 2008.

[92] H. Zhang, M. Dong, and H. Chen. Efficient and available in-memory KV-store with hybrid erasure coding and replication. In *FAST*, 2016.

[93] Z. Zhang, A. Deshpande, X. Ma, E. Thereska, and D. Narayanan. Does erasure coding have a role to play in my data center? Technical Report Microsoft Research MSR-TR-2010, 2010.

To Waffinity and Beyond: A Scalable Architecture for Incremental Parallelization of File System Code

Matthew Curtis-Maury, Vinay Devadas, Vania Fang, and Aditya Kulkarni
NetApp, Inc.
{mcm,vdevadas,vania,adityak}@netapp.com

Abstract

In order to achieve higher I/O throughput and better overall system performance, it is necessary for commercial storage systems to fully exploit the increasing core counts on modern systems. At the same time, legacy systems with millions of lines of code cannot simply be rewritten for improved scalability. In this paper, we describe the evolution of the multiprocessor software architecture (MP model) employed by the Netapp® Data ONTAP® WAFL® file system as a case study in incrementally scaling a production storage system.

The initial model is based on small-scale data partitioning, whereby user-file reads and writes to disjoint file regions are parallelized. This model is then extended with hierarchical data partitioning to manage concurrent accesses to important file system objects, thus benefiting additional workloads. Finally, we discuss a fine-grained lock-based MP model within the existing data-partitioned architecture to support workloads where data accesses do not map neatly to the predefined partitions. In these data partitioning and lock-based MP models, we have facilitated incremental advances in parallelism without a large-scale code rewrite, a major advantage in the multi-million line WAFL codebase. Our results show that we are able to increase CPU utilization by as much as 104% on a 20-core system, resulting in throughput gains of up to 130%. These results demonstrate the success of the proposed MP models in delivering scalable performance while balancing time-to-market requirements. The models presented can also inform scalable system redesign in other domains.

1 Introduction

To maintain a competitive advantage in the storage market, it is imperative for companies to provide cutting-edge platforms and software to maximize the returns from such systems. Recent technological trends have made this prospect more difficult, as increases in CPU clock speed have been abandoned in favor of increasing core counts. Thus, to achieve continuing performance gains, it has become necessary for storage systems to scale to an ever-higher number of cores. As one of the primary computational bottlenecks in storage systems, the file system itself must be designed for scalable processing.

Due to time-to-market objectives, it is simply not feasible to rewrite the entire code base of a production file system to use a new multiprocessor (MP) model. Reimplementing such a system to use explicit fine-grained locking would require massive code inspection and changes and would also carry with it the potential for introducing races, performance issues caused by locking overhead and contention, and the risk of deadlocks. In this paper, we present a series of techniques that have allowed us to simultaneously meet scalability and schedule requirements in the WAFL file system over the course of a decade and to minimize the code changes required for parallelization. In particular, all of our approaches emphasize *incremental parallelization* whereby common code paths can be optimized without having to make changes in less critical code paths. Although we evaluate these techniques in a file system, the approaches are not inherently limited to that context.

The first technique we discuss—referred to as *Classical Waffinity*—applied data partitioning to fixed-size regions of user files. This approach provided a mechanism to allow read and write operations to different ranges of user files to occur in parallel without requiring substantial code rewrite, because the use of data partitioning minimized the need for explicit locking. Extending this model, *Hierarchical Waffinity* further parallelized operations that modify systemwide data structures or metafiles, such as the creation and deletion of files, by implementing a hierarchical data partitioning infrastructure. Compared to Classical Waffinity, Hierarchical Waffinity improves core usage by up to 38% and achieves 95% average utilization across a range of critical workloads.

Finally, we have extended Hierarchical Waffinity to handle workloads that do not map neatly to these partitions by adding a fine-grained lock-based MP model *within* the existing data-partitioned architecture. This innovative model—called *Hybrid Waffinity*—provides significant parallelism benefits for previously problematic access patterns while not requiring any code changes for workloads where Hierarchical Waffinity already excelled. That is, the hybrid model introduces minimal explicit locking in narrowly defined cases to overcome specific scalability limitations in Hierarchical Waffinity to increase core usage by up to 104% and improve throughput by as much as 130%. Using these techniques has allowed WAFL to scale to the highest-end Data ONTAP platforms of their time (up to 20 cores) while constraining modifications to the code base.

The primary contributions of this paper are:

- We present a set of multiprocessor software architectures that facilitate incremental parallelization of large legacy code bases.
- We discuss the application of these techniques within a high-performance, commercial file system.
- We evaluate each of our approaches in the context of a real production storage system running a variety of realistic benchmarks.

Next, we present a short background on WAFL. Sections 3 through 5 present the evolution of the WAFL multiprocessor model through the various steps outlined above, and Section 6 evaluates each of the new models. Section 7 presents related work, and we conclude in Section 8.

2 Background on the WAFL File System

WAFL implements the core file system functionality within the Data ONTAP operating system. WAFL houses and exports multiple file systems called NetApp FlexVol® volumes from within a shared pool of storage called an *aggregate* and handles file system operations to them. In WAFL, all metadata and user data (including logical units in SAN protocols) is stored in files, called metafiles and user files, respectively. The file system is organized as a tree rooted at the super block. File system operations are dispatched to the WAFL subsystem in the form of messages, with payloads containing pertinent parameters. For detailed descriptions of WAFL, see Hitz et al. [17] and Edwards et al. [13].

Data ONTAP itself was first parallelized by dividing each subsystem into a private *domain*, where only a single thread from a given domain could execute at a time. For example, domains were created for RAID, networking, storage, file system (WAFL), and the protocols. Communication between domains used message passing. Domains were intentionally defined such that data sharing was rare between the threads of different domains, so this approach allowed scaling to multiple cores with minimal code rewrite, because little locking was required. The file system module executed on a dedicated set of threads in a single domain, such that only a single thread could run at a time. This simplistic model provided sufficient performance because systems at the time had very few cores (e.g., four), so parallelism within the file system was not important. Over time, each of these domains has become parallel, but in this paper we focus on the approaches used to parallelize WAFL.

3 Classical Waffinity

As core counts increased, serialized execution of file system operations became a scalability bottleneck because such operations represented a large fraction of the computational requirements in our system. To provide the initial parallelism in the file system, we implemented a multiprocessor model called *Waffinity* (for WAFL affinity), the first version of which was called *Classical Waffinity* and shipped with Data ONTAP 7.2 in 2006.

In Classical Waffinity, the file system message scheduler defined message execution contexts called *affinities*. User files were then partitioned into *file stripes* that corresponded to a contiguous range of blocks in the file (approximately 2MB in size), and these were rotated over a set of *Stripe affinities*. This model ensured that messages operating in different Stripe affinities were guaranteed to be working on different partitions of user files, so they could be safely executed in parallel by threads executing on different cores. In contrast, any two messages that were operating on the same region of a file would be enqueued within the same affinity and would therefore execute sequentially. This data partitioning provided an implicit coarse-grained synchronization that eliminated the need for explicit locking on partitioned objects, thereby greatly reducing the complexity of the programming model and the development cost of parallelizing the file system. Some locking was still required to protect shared global data structures that could be accessed by multiple affinities.

In Waffinity, we introduced a set of threads to execute the messages in each affinity, and we allowed the thread scheduler to simultaneously run multiple Waffinity threads. The Waffinity scheduler maintained a FIFO list of affinities with work; that is, affinities that had been sent messages that operated within that partition. Any running thread dynamically called into the Waffinity scheduler to be assigned an affinity from which to execute messages. The number of Stripe affinities was empirically tuned to be 12 and the number of Waffinity threads was defined per platform to scale linearly with the number of cores. NetApp storage systems at the time maxed out at 8 cores, which ensured more affinities than threads. Having more affinities than threads and creating a dynamic association between them decreased the likelihood of any thread being unable to find work.

The benefit of this model comes from the fact that most performance-critical messages at the time, such as user file reads and writes, could be executed in Stripe affinities, because they operated within a single user-file stripe. However, reads and writes across file stripes or operations such as Open and Close that touch file system metadata could not be performed in parallel from the Stripe affinities. To handle such cases, we provided a single-threaded execution context—called the Serial affinity—such that when it was scheduled, no Stripe affinities were scheduled and vice versa. This approach is analogous to a Reader-Writer Lock, where the Serial affinity behaves like a writer with exclusive file system access and the Stripe affinities behave like readers with shared access. Therefore, any messages requiring access to data that was unsafe from the Stripe affinities could be assured exclusive access to all file system data structures by executing in the Serial affinity. Use of the Serial affinity excessively serialized many operations; however, it allowed us to incrementally optimize the file system by parallelizing only those messages found to be performance critical. This approach also provided an option whereby unsafe code paths could be dynamically re-sent to the Serial affinity (called a *slowpath*).

Classical Waffinity exposed sufficient concurrency to exploit high-end NetApp platform core counts at the time, i.e., 8 cores. That is, further parallelizing file system operations would not have had a major impact on overall performance because the cores were already well used. However, as the number of cores increased, limitations in the partitioning provided by the model led to scalability bottlenecks. For example, an SFS2008 benchmark running on 12 cores saw the Serial affinity in use 48% of the time, as a result of the metadata operations such as Setattr, Create, and Remove. This serialization resulted in considerable core idle time, as evaluated in Section 6.1.1, which reduced potential performance. Further, most of the work remaining could not be moved to a Stripe affinity due to the strict rules of what can run there (i.e., operating within a single user file stripe). With higher core counts, serialized execution had a larger impact on performance, because the speedup achieved through parallelism was limited by the serial time, in accordance with Amdahl's law. Thus, although sufficient at the time, Classical Waffinity as first designed was simply unable to provide the required performance going forward.

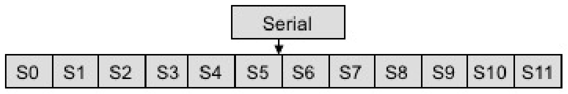


Figure 1: The structure of Classical Waffinity with the Serial affinity on top and the twelve Stripe affinities below.

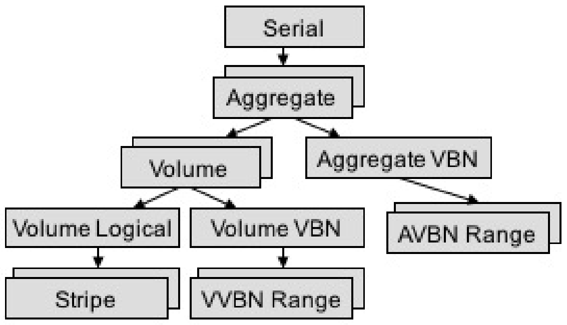


Figure 2: The Hierarchical Waffinity hierarchy rooted at the Serial affinity.

4 Hierarchical Waffinity

Hierarchical Waffinity builds on top of the Classical Waffinity model to enable increased levels of parallelism. This model, which first shipped with Data ONTAP 8.1 in 2011, greatly reduces the increasingly critical single-threaded path by providing a way to parallelize additional work. Further, the model offers insight into protecting hierarchically structured systems in other domains [22] by using data partitioning.

4.1 The Affinity Hierarchy

An alternative view of the Serial and Stripe affinities of Classical Waffinity is as a hierarchy, as shown in Figure 1, where a node is mutually exclusive with its children but can run in parallel with other nodes at the same level. Hierarchical Waffinity facilitates additional parallelism by extending this simple 2-level hierarchy to efficiently coordinate concurrent accesses to other fundamental file system objects beyond data blocks of user files.

Each affinity is associated with certain permissions, such as access to metadata files, that serialized execution in the file system in Classical Waffinity (Figure 2 and Table 1). The new affinity scheduler enforces execution exclusivity between a given affinity and its children, so Hierarchical Waffinity only restricts the execution of an affinity's *ancestors* and *descendants* (hierarchy parents and children, respectively); all other affinities can safely run in parallel. For example, if the Volume Logical affinity is running, then its Stripe affinities are excluded along with its parent Volume, Aggregate, and Serial affinities.

This design ensures that no two messages with conflicting data accesses run concurrently, because they would run in affinities that exclude one another. Hierarchical Waffinity is analogous to a hierarchy of Reader-Writer Locks, where running in any affinity acquires the lock as a writer and thus prevents any readers (i.e., descendants) from running concurrently, and vice versa.

The WAFL file system is itself hierarchical (i.e., buffers within inodes within FlexVol volumes within aggregates), making Hierarchical Waffinity a natural fit. Knowledge of the specific data access patterns that are most common in WAFL informed the decision of affinity layout in the hierarchy and the mapping of specific object accesses to those affinities.

As a general rule, client-facing data, such as user files, logical units, and directories, is mapped to the Volume Logical branch of the hierarchy and internal metadata is mapped to Volume VBN—so named because it is typically indexed by volume block number (VBN). This allows parallelism between client-facing and internal operations within a single volume. The Aggregate, Volume, Stripe, and Range affinity types have multiple *instances*, which allows parallel execution of messages operating on disjoint data, such as any two operations in different aggregates, FlexVol volumes, or regions of blocks in a file. Each file system object is mapped to a particular instance in the affinity hierarchy, based on its location in the file system. The number of instances of each affinity as well as the mapping of objects to instances can be adapted to the observed workload to maximize performance. Further, new affinity types can be (and have been) added to the hierarchy over time in response to new workloads and data access patterns.

4.2 Affinity Access Rules

The affinity permissions required by a message are determined by the type of objects being accessed and the access types. We used knowledge of the system to derive affinity permissions that allowed performance-critical operations to run with the most parallelism. In WAFL, any access is associated with a specific affinity, using the rules shown in Table 1. *Exclusive* access to an object ensures that no concurrently running affinities can access that object. The fundamental objects that are protected by data partitioning in Waffinity are buffers, files, FlexVol volumes, and aggregates. Accesses to other object types are infrequent but are protected with fine-grained locking, and deadlock is prevented through the use of lock hierarchies.

Each FlexVol volume and aggregate is mapped to a Volume and Aggregate affinity when it comes online. In the WAFL file system, files are represented by *inodes*, which are mapped to an affinity within the hierarchy of the volume or aggregate in which they reside. Inodes can be accessed in either *exclusive* or *shared* mode. In exclusive mode, only one message has access to the inode and consequently can change any inode property or free the inode. In shared mode, the inode's fundamental properties remain read-only, but the majority of an inode's fields can be modified and are protected from concurrent accesses by fine-grained locking. Similar distinctions are in place for FlexVol volumes and aggregates.

Blocks are represented in memory by a buffer header and a 4KB payload. Details of the WAFL buffer cache architecture have previously been published [11]. Accesses to buffers fall into the following four categories:

- Insert: A new buffer object is allocated and the payload is read in from disk. The buffer becomes associated with the corresponding inode.
- Read: The payload of an in-memory buffer is read. No disk I/O is required in this case.
- Write: The payload of the in-memory buffer is modified in memory. The buffer header is also modified to indicate that it is dirty.
- Eject: The buffer is evicted from memory.

Each of these access modes is mapped to a specific affinity instance for a given buffer. For Write, Insert, and Eject accesses, our data partitioning model requires that only a single operation perform any of these accesses at a time. Thus, operations must run in the designated affinity for a given access mode or its ancestor. On the other hand, read accesses are safe in parallel with each other, but it is necessary to ensure that the buffer will not be ejected underneath it, so that it can run in any ancestor or descendant of the Eject affinity. Typically, Write and Eject access are equivalent, which similarly prevents concurrent reads and writes. The affinity mappings for buffers are chosen to allow maximum parallelism, while ensuring access to the buffers from all necessary affinities. For example, user file reads and writes run in the Stripe affinities because the relevant buffers map to these affinities.

4.3 Mapping Operations to Affinities

Messages are sent to a predetermined affinity based on the required permissions of the operation, as identified by the software developer. If a message requires the permissions assigned to multiple affinities, then it is directed to an affinity that is an ancestor of each. For example, an operation that requires privileges assigned to a particular

Affinity Name	Access Privileges Provided
Stripe	Provides exclusive access to a predefined, distinct set of blocks. Enables concurrent access to sub-file-level user data.
Volume Logical	Provides exclusive access to most client-visible files (and directories) within a volume and to certain file system internal metadata.
Volume VBN	Provides exclusive access to most file system metadata, such as those files that track block usage. Such files are typically indexed by volume block number (VBN).
Volume VBN Range	Provides access to specific ranges of blocks within files belonging to Volume VBN.
Volume	Provides exclusive access to all files within a volume as well as per-volume metadata that lives in the containing aggregate.
Aggregate VBN	Similarly to Volume VBN, provides access to most file system metadata in an aggregate.
Aggregate VBN Range	Provides access to specific ranges of blocks within files belonging to Aggregate VBN.
Aggregate	Provides exclusive access to files in an aggregate.
Serial	Inherits the access rights of all other affinities and provides access to other global data.

Table 1: The affinities and the access rights that they provide.

Stripe affinity and to a Volume VBN affinity could safely execute in the Volume affinity. As in Classical Waffinity, if a message is routed to a particular affinity and later determines that it requires additional permissions, it may be dynamically re-sent (i.e., slowpath) to a *coarser* (i.e., less parallel) affinity that provides the necessary access rights. Object accesses are achieved through a limited set of APIs that we have updated to enforce the required affinity rules, including slowpathing if necessary. Thus, the programming model helps ensure code correctness. The more access rights required by a message, the higher in the affinity hierarchy it must run and the more it limits its parallelism by excluding a larger number of affinities from executing. Thus, it is preferable to run in as *fine* an affinity as possible. For this reason, we have selected data mappings that allow the most common operations to be mapped to fine affinities, and objects that are frequently accessed together are given similar mappings.

Figure 3 shows several example affinity mappings of operations under a single Volume affinity. For simplicity, we assume a file stripe size of 100 contiguous user blocks, or 10 blocks in metafiles. Reads and writes within a file stripe map to a Stripe affinity. However, a user file deletion ("Remove: A") runs in the Volume Logical affinity, because it requires exclusive access to that user file, as does a write operation that spans multiple file stripes ("W: A[100..200]"). Running in this affinity prevents the execution of any reads or writes to blocks in that file. However, reads and writes to files in other volumes are not impacted, nor are operations on file system metadata within the same volume. Note that "W: A, MD" must run in Volume affinity to write to both a user file and metafile. As noted earlier, key message parameters are tracked in the message payload. When a message is sent into WAFL, its payload is inspected to determine the type of the message and other data from which the destination affinity is calculated. Effectively, each message exposes the details of its data accesses to the scheduler so that MP safety can be enforced at this level, similar to some language constructs for task-based systems [3, 34]. For example, a write message exposes the offsets being written and thus the required affinity for execution.

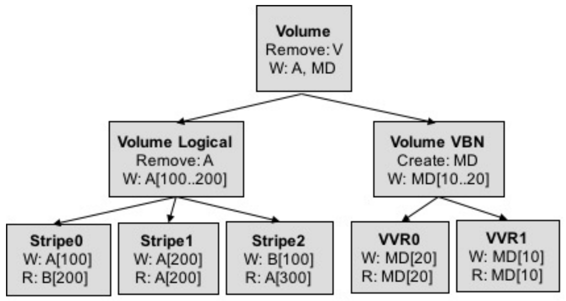


Figure 3: Example affinity mappings of Read ("R"), Write ("W"), Remove, and Create operations to user files A and B, metadata file MD, and FlexVol V. A block offset of 100 with file A is denoted as A[100].

4.4 Development Experience with Hierarchical Waffinity

Hierarchical Waffinity allows parallelization at the granularity of a message type, running all other message types in the Serial affinity, allowing *incremental optimization* over time. Thus, parallelization effort scales with the size of the message, rather than requiring the entire code base to be updated at once. A typical message handler is on the order of *hundreds* or *thousands* of lines of code, whereas the file system is on the order of *millions*. Further, a message can first be parallelized into one affinity and later moved to a finer affinity when need arises and the extra development cost can be justified.

Messages often require only minimal code changes during parallelization because data access guarantees are provided by the model. In such cases, after a detailed line-by-line code inspection to evaluate multiprocessor safety, messages can be moved into fine affinities just by changing the routing logic that computes the target affinity. For example, parallelizing the Link operation to run in the Volume Logical affinity required fewer than 20 lines of code to be written, none of which were explicit synchronization. In other cases, major changes are required to safely operate within a single affinity, for example by restructuring data accesses, thus requiring potentially thousands of lines of code changes. In WAFL, the underlying infrastructure required to implement the affinities, scheduling, and rule enforcement amounted to approximately 22K lines of code. Compared to the alternative of migrating the whole file system to fine-grained locking, which would involve inspecting and updating a large fraction of the millions of lines of code, these costs are relatively small.

Software systems in many domains employ hierarchical data structures (such as linear algebra [12] and computational electromagnetics [14]), and a variety of techniques have been developed to provide multiprocessor safety in such cases [15, 22]. Hierarchical Waffinity offers an alternative architecture that is capable of incremental parallelization. In applying this approach to other systems, the types of affinities to create, the number of instances of each type, and the mapping of objects to affinities would be based on domain-specific knowledge of the data access patterns. In practice, this approach applies most naturally to message passing systems where a subset of message handlers could be parallelized while leaving others serialized, or alternatively to task-based systems such as Cilk++ [26].

4.5 Hierarchical Scheduler

The Hierarchical Waffinity scheduler is an extension of the Classical Waffinity scheduler that maps the now greater set of runnable affinities to the Waffinity threads for execution while enforcing the hierarchical exclusion rules. As before, Waffinity threads are exposed to the general CPU scheduler, and when they are selected for execution, they begin by calling into the Waffinity scheduler for work. A running thread then begins processing the messages queued up to that affinity for the duration of an assigned quantum, after which the thread calls back into the scheduler to request another affinity to run.

As messages are sent into WAFL and processed by the Waffinity threads, the Waffinity scheduler tracks the sets of affinities that are runnable (i.e., with work and not excluded), running, or excluded by other executing affinities. When threads request work, the scheduler selects an affinity from the runnable list, assigns it to the thread, and updates the scheduler state to reflect the newly running affinity. In particular, the scheduler maintains a queue of affinities that is walked in FIFO order to find an affinity that is not excluded. An excess of work in coarse affinities manifests in the scheduler as a shortage of runnable affinities, resulting in situations where available threads cannot find work to do and must sit idle, which in turn results in wasted CPU cycles. We also track the number of runnable affinities and ensure that the optimal number of threads are in flight any time an affinity begins or ends execution. To prevent the starvation of coarse affinities, we periodically drain all running affinities and ensure that all affinities run with some regularity. Figure 4 shows a sample scheduler state. In this example, there are seven affinities currently running and five more that can be selected for execution by the affinity scheduler.

4.6 Waffinity Limitations and Alternatives

Fundamental in the Waffinity architecture is a mapping of file system objects to a finite set of affinities, often causing independent operations to unnecessarily become serialized. For example, any two operations that require exclusive access to two user files in the same volume (such as deletion) are serialized. As long as sufficient parallelism is found to exploit available cores on the target platforms, this limitation is acceptable in the sense that increasing available parallelism will not result in increased performance. Two other scenarios that can result in significant performance loss are 1) when frequently accessed objects directly map to a coarse affinity; and 2) when two objects mapped to different affinities must be accessed by the same operation. These scenarios are not well handled in Hierarchical Waffinity and result in poor scaling in several important workloads, as shown in Section 6.

An alternative to the Waffinity architecture would be to use fine-grained locking to provide MP safety. In such an approach, all file system objects would be protected by using traditional locking, and no limits need to be imposed on which operations can be executed in parallel. This would provide additional flexibility in the programming model; however, it carries many drawbacks

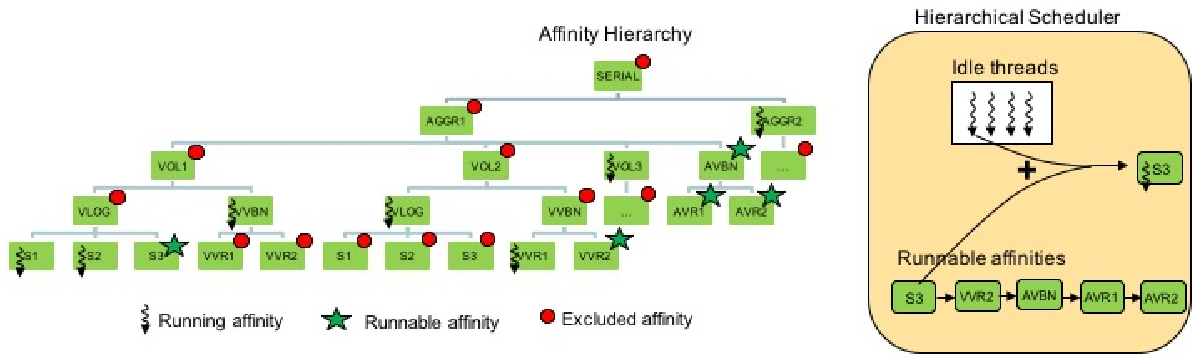


Figure 4: Sample hierarchical affinity scheduler state.

that led to the decision to implement Classical and Hierarchical Waffinity. Reimplementing the file system to exclusively use fine-grained locks would require a massive code rewrite and would carry with it the potential for introducing races, performance issues caused by locking overhead and contention, and the risk of deadlocks. Instead, our approach reduces the overall amount of locking required, and even allows messages to run in coarse affinities without locking.

5 Hybrid Waffinity

The next step in the multiprocessor evolution of WAFL is *Hybrid Waffinity*, which shipped with Data ONTAP 9.0 in 2016. This model is designed to scale workloads for which Hierarchical Waffinity is not well suited, due to a poor mapping of data accesses to affinities, as discussed in Section 4.6. This model supports fine-grained locking *within* the existing hierarchical data partitioned architecture to protect particular objects when accesses do not map neatly to fine partitions. At the same time, we continue to use partitioning where it already excels, such as for user file reads and writes. Overall, Hybrid Waffinity leverages both fine-grained locking *and* data partitioning in cases where each approach excels. Although this may seem like an about-face from the data partitioned models, in fact it is merely an acknowledgement that in some cases fine-grained locking is required for effective scaling. Retaining partition-based protection in most cases is critical so that only the code that scales poorly with data partitioning needs to be updated.

In Hybrid Waffinity, we allow buffers in a few select metafiles to be protected by locking, while the vast majority of buffers, as well as all other file system objects, continue to use data partitioning for protection. The result is a *hybrid* model of MP-safety where different buffer types have different protection mechanisms. The use of locking allows these buffers to be accessed from finer affinities; however, we continue to allow *lock-free* access from a coarser affinity. That is, because all object accesses occur within some affinity subtree, a message running in the root of that subtree can safely access the object without locking.

5.1 Hybrid-Insert

With Hierarchical Waffinity, each buffer is associated with a specific Insert affinity that protects the steps involved in inserting that buffer. This mandates that all messages working on inserting the buffer will run in the same affinity to be serialized. This design is effective in the common case where a single buffer is accessed or multiple buffers with similar affinity mappings are accessed. However, in cases where a message accesses multiple buffers in different partitions, the message must run in a coarser affinity that provides all of the necessary permissions. Figure 5 illustrates a scenario in which both User file buffers and Metafile buffers must be accessed, which happens when replicating a FlexVol volume (discussed in Section 6.2.3). This operation must run in the AGGR1 affinity, rather than the more parallel S1 or AVR1 affinity. In such cases, parallelism in the system is reduced, because time spent running in coarse affinities limits the available affinities that can be run concurrently, potentially starving threads and cores of work.

We overcome these shortcomings with Insert by allowing *Hybrid-Insert* access to certain buffers from *multiple* fine affinities. Only a few buffer types are frequently accessed in tandem with other buffers, and we apply Hybrid-Insert only in such cases. Thus, a message accessing two buffers now runs in the traditional (fine) Insert affinity of one buffer and protects the second buffer by using Hybrid-Insert, rather than in a coarse affinity with Insert access to both buffers. Allowing multiple affinities to insert a buffer means that two messages can simultaneously insert the same buffer, but Hybrid-Insert resolves such races and synchronizes all callers of any

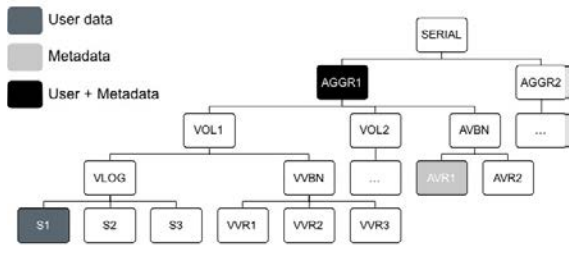


Figure 5: Hierarchical Waffinity model with User data access in S1 and Metadata access in AVR1.

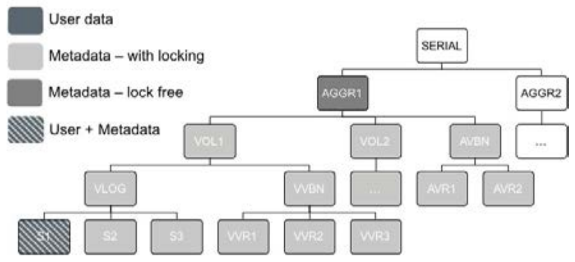


Figure 6: Hybrid Waffinity model where User data access continues to be in S1, but Metadata access is permitted from any descendant of AGGR1.

insert code path. Referring back to Figure 5, the User data can continue to be protected with partitioning in S1 and the Metadata can now be accessed from the S1 affinity (for example) with explicit synchronization, as shown in Figure 6. Further, noncritical messages that operate on Metadata can run in AGGR1 without being rewritten, because the scheduler will not run any other messages that access this data.

For explicit synchronization, Hybrid-Insert uses what we refer to as *MP-barriers*. MP-barriers employ a set of spin-locks that are hashed based on buffer properties. The insert process consists of a series of critical sections of varying lengths. Short critical sections can simply hold the spin-lock for their duration. However, longer critical sections avoid holding the lock for long periods by instead stamping the buffer with an *in-progress* flag under the lock at the beginning of the critical section and clearing it at the end. Other messages that encounter the in-progress flag can then block, knowing that a message is already moving this buffer toward insertion.

5.2 Hybrid-Write

The next buffer access mode we consider is Write, which involves changing the contents of a buffer and updating associated metadata. Hierarchical Waffinity serializes all readers and writers to the same buffer by mapping all such operations to the same affinity. Thus, Write access made it safe for writers to modify the buffer without locking and for readers to know that no writes were happening concurrently to the buffer. However, as with Insert access, messages requiring access to multiple buffers with different affinity mappings needed to run in a coarse affinity, thereby limiting parallelism.

Hybrid-Write allows writes to certain types of buffers from multiple affinities so that messages routed to an affinity for Write access to one buffer will also be able to access a different buffer by using Hybrid-Write, again as in Figure 6. Thus, readers and writers must synchronize by using fine-grained locking to ensure MP-safety and data consistency. In the new model, we have retained the traditional Write affinity in which an operation can run without locking. Read access has been redefined for Hybrid-Write buffer types such that it now maps to the traditional Write affinity, thereby providing (slow) read access to the buffer without any locking. New access modes called *Shared-Write* and *Shared-Read* have been added to provide access from finer affinities, and only by explicitly using these access modes—and thus implicitly agreeing to add the necessary locking—is any additional parallelism achieved. Thus, Hybrid-Write uses an *opt-in* model wherein legacy code remains correct by default until it is manually optimized.

Hybrid-Write uses spin-locks to protect buffer data and metadata. Spin-locks are sufficient here because the critical sections for reads and writes are typically small. The introduction of fine-grained locking increases complexity; however, it can be done incrementally as required by specific messages without a large-scale code rewrite. As an example of the increased complexity, buffer state observed at any time in Hierarchical Waffinity could always be trusted because the entire message execution was under the implicit locking of the scheduler. In contrast, buffer state observed inside an explicit critical section can no longer be trusted once the lock is released.

5.3 Hybrid-Eject

The final buffer access mode is Eject, which provides exclusive access and allows arbitrary updates to the buffer, including evicting the buffer from memory. Thus, Eject access maps to an affinity that excludes all affinities with access to this buffer. *Hybrid-Eject* instead uses fine-grained locking to provide exclusive access from a finer affinity. Unlike Hybrid-Insert and Hybrid-Write, we compute a *single* Hybrid-Eject affinity for each buffer to serialize all code paths. While Hybrid-Eject could always be used in place of Hybrid-Write, the semantics of Hybrid-Eject are more restrictive and would limit perfor-

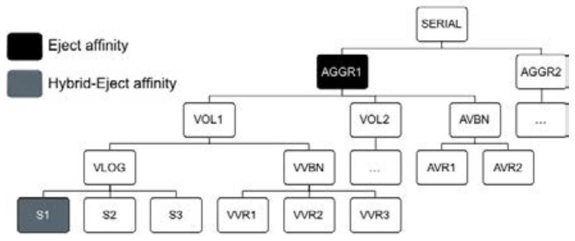


Figure 7: Waffinity model where traditional Eject access maps to AGGR1, but Hybrid-Eject maps to a specific fine affinity S1.

mance. We have also retained the traditional Eject affinity to minimize required code changes. Figure 7 shows a sample hierarchy with Eject affinity in AGGR1 and Hybrid-Eject in S1.

Simply protecting each buffer with a spin-lock would not be feasible for Hybrid-Eject because messages can read many buffers, each of which must be protected from ejection. Instead, we track a global serialization count that is incremented at periodic serialization points within the file system. Whenever a reference to a buffer is taken, it is stamped with the current serialization count under a spin-lock and is said to have an *active stamp*. Buffers with active stamps cannot be evicted and are implicitly unlocked when the serialization count is next incremented. Because message execution cannot span serialization points, buffers with stale stamps can be safely ejected. Preventing the ejection of a buffer for this duration is excessive but practical, since only 0.002% of buffers considered for ejection had an active stamp during an experiment with heavy load on a high-end platform.

To extend this infrastructure beyond buffer ejection, we also define an *exclusive stamp* that prevents any concurrent access at all. A message requiring exclusive access can simply put this value on the buffer and all subsequent accesses to the buffer spin until the exclusive stamp has been cleared. Spinning is acceptable in practice, since only 0.007% of exclusive stamp attempts encountered an active stamp in an experiment with heavy load.

5.4 Development Experience with Hybrid Waffinity

Adopting Hybrid Waffinity within WAFL involved creating the underlying infrastructure and then parallelizing individual messages to take advantage of it. For each of the access modes, we required approximately 3K lines of code changes, which included extensive rule enforcement and checking. As in the case of Hierarchical Waffinity, the effort involved in each specific message parallelization varied widely. Leveraging Hybrid-Insert and Hybrid-Eject requires few code changes because the infrastructure is primarily embedded within existing APIs. Hybrid-Write, on the other hand, requires more code changes due to the addition of fine-grained locking throughout the message handler. For example, all three messages that we optimized using Hybrid-Eject required fewer than 20 lines of code changes. In contrast, two messages parallelized using Hybrid-Insert and Hybrid-Write required a few thousand lines of changes.

We have already begun to apply this technique to other objects within WAFL. In particular, a project is under way to further parallelize access to certain inodes in WAFL by using Hybrid Waffinity, and we have found the code changes to be relatively modest. We are optimistic that the ease with which this technique was applied to inodes will translate to other software systems. The techniques of Hybrid Waffinity can potentially be applied in any data partitioned system, not only those that are hierarchically arranged. Prior work has discussed the difficulty in operating on data from different partitions [21, 38], and our approaches can be used in such cases to improve parallelism. For example, consider a scientific code operating on two arrays. Tasks could be divided up based on a partitioning of one array, and access to the other array could be protected through fine-grained locking.

6 Performance Analysis

6.1 Hierarchical Waffinity Evaluation

To highlight the improvements provided by Hierarchical Waffinity, we chose two performance benchmarks that emphasize the limitations of Classical Waffinity. Hierarchical Waffinity was released in 2011 as part of Data ONTAP 8.1, and we used this software to evaluate its benefits. In this section, the benchmarks were run on a 12-core experimental platform that was the highest-end Data ONTAP platform available at the time this feature shipped. Many changes were made in Data ONTAP between releases, so we cannot directly compare the approaches. Instead, we used an instrumented kernel that runs messages in the same affinities as would Classical Waffinity for our baseline to isolate the impact of our changes. We used multiple FlexVol volumes in these experiments because this is representative of the majority of customer setups.

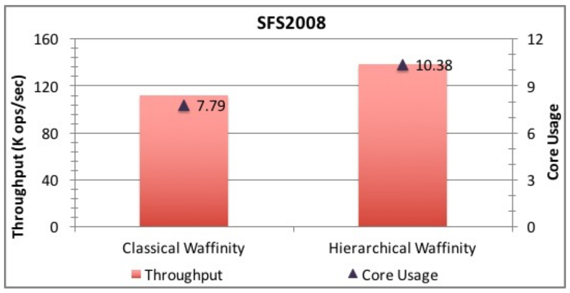


Figure 8: Throughput and core usage improvements with Hierarchical Waffinity compared to Classical Waffinity on a Spec SFS2008 workload.

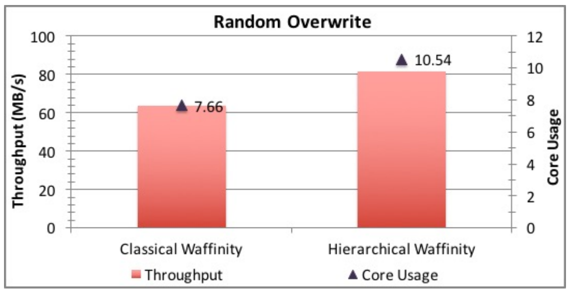


Figure 9: Throughput and core usage improvements with Hierarchical Waffinity compared to Classical Waffinity on a random overwrite workload.

6.1.1 Spec SFS2008

We first measured performance using the Spec SFS2008 benchmark [35] using NFSv3 on 64 FlexVol volumes. This workload generates mixed workloads that simulate a "typical" file server, including Read, Write, Getattr, Lookup, Readdir, Create, Remove, and Setattr operations. Several of these operations involve modification of file attributes, so the corresponding messages were forced to run in the Serial affinity in the Classical Waffinity model. Hierarchical Waffinity allowed us to move Create and Setattr into the Volume Logical affinity and Remove to the Volume affinity. Since there are eight Volume affinities, up to eight Create/Setattr/Remove operations can now run in parallel with each other and can also run in parallel with the remaining client operations in Stripe affinities.

Figure 8 shows the throughput and core usage improvements of Hierarchical Waffinity over Classical Waffinity for SFS2008. As noted, in Classical Waffinity many operations ran in the Serial affinity, thereby serializing all file system operations and resulting in idle cores. In the baseline, the Serial affinity was busy 48% of the time, thus limiting parallel execution to only 52% of the time. In contrast, by providing additional levels of parallelism, the hierarchical model was able to reduce Serial affinity usage to 9%. Alleviating this significant scalability bottleneck increased the system-wide core usage by 2.59 out of 12 cores, showing that parallelism is significantly improved through the ability to process operations on metadata in parallel with each other and with reads and writes. Most importantly, the additional core usage successfully translated into a 23% increase in throughput. That is, Hierarchical Waffinity effectively exploits the additional processing bandwidth to improve performance.

6.1.2 Random Overwrite

We next evaluate the benefit of Hierarchical Waffinity on a 64 FlexVol volume random overwrite workload. Block overwrites in the file system are particularly interesting because WAFL always writes data to new blocks on disk. Thus, for each block overwritten, the previously used block on disk must be freed and the corresponding file system metadata tracking block usage must be updated. It is not the write operation itself that is of interest in this experiment, because that was parallelized even in Classical Waffinity. This benchmark instead demonstrates the gains from parallelizing block free operations that used to run in the Serial affinity, because they involve updating file system metadata. With Hierarchical Waffinity, these messages can now run in the Volume VBN and Aggregate VBN affinities (when freeing in a FlexVol volume and aggregate, respectively) because these affinities provide access to all of the required metafiles. Thus, the changes provided in Hierarchical Waffinity 1) allow block free messages to run concurrently with each other on different FlexVol volumes because each volume has its own Volume VBN affinity; and 2) allow block free messages to run in parallel with client operations in the Stripe affinities.

Here again, Hierarchical Waffinity demonstrates a significant reduction in Serial affinity usage, from 27% to 7%, as a result of parallelizing the block free operations. This reduction in serialization made it possible for the same workload to scale to an additional 2.88 cores compared to Classical Waffinity, as shown in Figure 9. This extra core usage allowed an overall improvement in benchmark throughput of 28%. The random write workload is interesting because it demonstrates the benefits of running internally generated metadata operations in Volume VBN affinity in parallel with front-end client traffic in the Stripe affinities, which was not possible in Classical Waffinity.

6.1.3 Overall CPU Scaling

The above analysis of Hierarchical Waffinity showed a substantial improvement in the number of cores used, but 1.5 cores were still idle. These benchmarks were se-

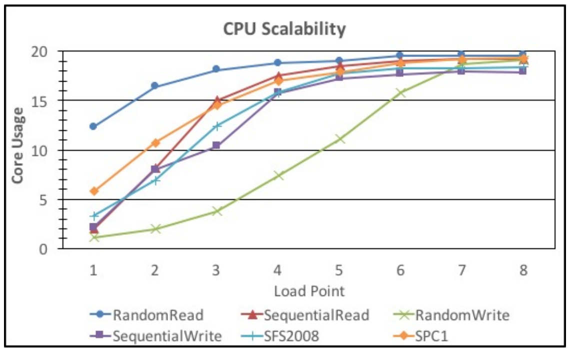


Figure 10: Core usage with Hierarchical Waffinity on a variety of workloads under increasing levels of load.

lected to illustrate the benefits of Hierarchical Waffinity *compared to* Classical Waffinity, not maximum utilization. Idle cores could have been driven down still further by parallelizing even the remaining 9% of Serial execution in SFS2008 and 7% in random write, but this was not required to meet performance and scaling objectives at the time.

In subsequent releases, we have further parallelized many operations, resulting in the improved CPU scalability shown in Figure 10. These parallelization efforts have included both reducing Serial affinity usage and moving already parallelized work into still finer affinities. In particular, the graph shows the achieved core usage at increasing levels of load (i.e., load points) for a variety of workloads on 64 FlexVol volumes on a 20-core storage server running Data ONTAP 9.0. The key takeaways are the low core usage that occurs at low load and the high core usage achieved in the presence of high load. In particular, four of the six benchmarks achieve a utilization of 19+ cores, with all benchmarks reaching at least 18 cores. This data demonstrates that Hierarchical Waffinity is able take advantage of computational bandwidth up to 20 cores in a broad spectrum of important workloads, and cores are not starved for work by an excess of computation in coarse affinities. In Section 6.3, we discuss the issue of continued scaling on future platforms.

6.2 Hybrid Waffinity Evaluation

Although the previous section demonstrated the success of Hierarchical Waffinity across a wide range of workloads, there are also cases where its scalability is limited. Thus, we next evaluate the benefits of the Hybrid Waffinity model by considering benchmarks that emphasize cases where the hierarchical model falls short. We focus on single FlexVol volume scenarios because any coarse affinity utilization significantly limits parallelism; however, we also consider the scalability with multiple FlexVol volumes where Hierarchical Waffinity itself is typically very effective already. Single-volume configurations are less common, but they are still a very important customer setup. This section describes analysis done with Data ONTAP 9.0 on a 20-core platform, the highest-end system available in 2016.

6.2.1 Sequential Overwrite

We first look at the benefits of Hybrid Waffinity on a sequential overwrite workload. As discussed above, block overwrites are interesting in WAFL because they result in block frees. In Hierarchical Waffinity, block free work runs in the Volume VBN affinity, because it requires updating various file system metadata files, so it already runs in parallel with front-end traffic in the Stripe affinities. However, if the system is unable to keep up with the block free work being created, then client operations must perform part of the block free work, which hurts performance. This problem is exacerbated on single-volume configurations where all block free work in the system must go through the single active Volume VBN affinity, which can become a major bottleneck.

Increasing the parallelism of this workload requires running block free operations in Volume VBN Range (or simply *Range*) affinities. Certain metafile buffers required for tracking free blocks can be mapped to specific Range affinities, but other metafiles need to be updated from any Range affinity, so the operation must run in an affinity that excludes both relevant affinities. Fortunately, this is an ideal scenario for *Hybrid-Insert* and *Hybrid-Write*, in that buffers from certain metafiles can be mapped to a specific Range affinity and others can be protected by using locking from any Range affinity. Using a combination of partitioning and fine-grained locking to protect its buffer accesses, Hybrid Waffinity allows the block free operation to run in 1) a finer affinity and 2) an affinity of which there are multiple instances per FlexVol volume.

We evaluate a single-volume sequential overwrite benchmark on a 20-core platform with all flash drives. Figure 11 shows the throughput achieved and core usage at increasing levels of sustained load from a set of clients (i.e., the load point). Comparing peak load points shows a 62.8% improvement in throughput from the use of 4.8 additional cores. Under sufficient load, the Hierarchical Waffinity performance falls off, because block free work is unable to keep pace with the client traffic generating the frees. Hybrid Waffinity prevents this from happening by increasing the computational bandwidth that can be applied to block free work on a single FlexVol volume. This experiment demonstrates scalability up to 13.2 cores; however, repeating the test with 64 volumes

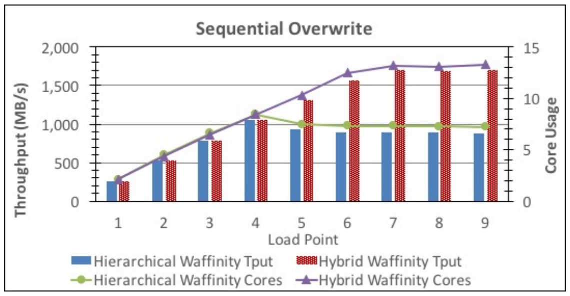


Figure 11: Throughput and core usage improvements at various levels of server load with Hybrid Waffinity compared to Hierarchical Waffinity for a sequential overwrite workload.

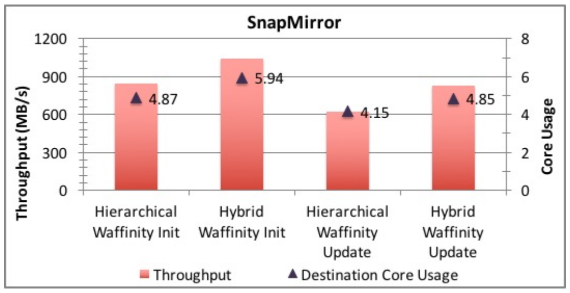


Figure 12: Throughput and core usage improvements with Hybrid Waffinity compared to Hierarchical Waffinity for a SnapMirror workload. Core usage is out of 20 available cores.

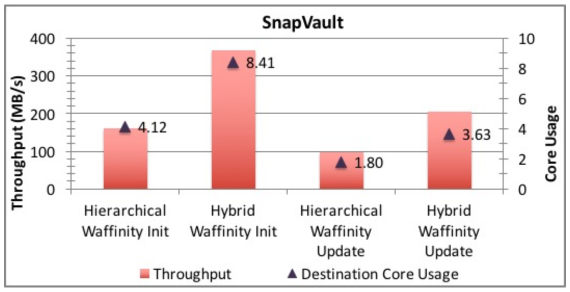


Figure 13: Throughput and core usage improvements with Hybrid Waffinity compared to Hierarchical Waffinity for a SnapVault workload. Core usage is out of 20 available cores.

shows the system scaling further to 16.6 cores due to the activation of additional Volume affinity hierarchies. Hybrid Waffinity benefits throughput by only 7.5% and core usage by 1.6 cores in the multi-volume case because Hierarchical Waffinity is already so effective.

6.2.2 NetApp SnapMirror

The next workload we consider is NetApp SnapMirror®, a technology that replicates the contents of a FlexVol volume to a remote Data ONTAP storage server for data protection [33]. During the "init" phase, the entire contents of the volume are transferred to the destination and subsequent "update" transfers replicate data that has changed since the last transfer. In particular, SnapMirror operates by loading the blocks of a metadata file representing the entire contents of the volume on the source, sending the data to a remote storage server, and writing to the same metafile on the destination volume. Exclusive access to this file's buffers belongs to the Volume affinity, which results in substantial serialization, because such work serializes all processing within the volume. *Hybrid-Eject* allows these buffers to instead be processed in the Stripe affinities.

Figure 12 shows the benefits of Hybrid-Eject on a single-FlexVol SnapMirror transfer. The transfer is destination-limited, so we evaluate core usage on the destination. During the init phase, core usage goes up by 1.1 cores and the throughput is improved by 24.2%. Similarly, the update phase uses an additional 0.7 cores, resulting in a 32.4% gain in throughput. Despite the benefit, core usage is low in the single-volume case; however, an experiment replicating 24 volumes scales to 12.5 cores (init) and 9.8 cores (update), at which point the workload becomes bottlenecked elsewhere and Hybrid Waffinity provides no benefit beyond the hierarchical model alone.

6.2.3 NetApp SnapVault

We conclude our analysis by looking at the performance benefits of *all three hybrid access modes* together. Another Data ONTAP technology for replicating FlexVol volumes, called SnapVault®, writes to both user files *and* metafiles on the destination server. In Hierarchical Waffinity, the operations run in a coarse affinity (such as Volume or Volume Logical) with access to both types of buffers. With Hybrid Waffinity, user file buffers remain mapped to Stripe affinities, but Hybrid-Write and Hybrid-Insert allow the metafile buffers to be accessed by any child of the Volume affinity. Thus, SnapVault operations can run in the Stripe affinity of their user file accesses and use locking to protect metadata accesses. Further, the metafile buffers map to Volume Logical for Eject access, which can be optimized by using Hybrid-Eject to facilitate processing in the Stripe affinities.

Figure 13 presents the improvements in throughput and core usage of Hybrid Waffinity on a single-volume SnapVault transfer. The new model facilitated a throughput gain of 130% in the init phase on an extra 4.29 cores used out of 20 available. Update performance is similarly improved by 112% on a core usage increase of 1.83 cores. Increasing the transfer to 8 volumes increases the total core usage to 17.7 cores (init) and 10.6 cores (update), at which point the primary bottlenecks move to other subsystems. Even here, the hybrid model improves scalability by 0.5 cores and 3.1 cores and throughput by 20.2% and 18.1%, for init and update, respectively.

In summary, Hybrid Waffinity is able to greatly improve performance in single-volume scenarios where Hierarchical Waffinity struggles most, and even improves scaling in certain multi-volume workloads.

6.3 Discussion of Future Scaling

The analysis described in this paper was conducted on the highest-end platforms available at the time each feature was released. These platforms drive the scalability investment that is made, because scaling beyond available cores does not add customer value. New platforms with higher core counts will continue to enter the market in the future and our requirements will continue to increase as a result. We expect that the infrastructure now in place will continue to pay rich dividends as future parallelization investments focus on utilizing the techniques discussed in this paper to greater degrees rather than inventing new ones. That is, the bottlenecks on the horizon are not a limitation of the architecture itself. Internal systems with more cores are currently undergoing extensive tuning, and the techniques discussed in this paper have already allowed scaling well beyond 30 cores.

7 Related Work

Operating system scalability for multicore systems has been the subject of extensive research. Recent work has emphasized minimizing the use of shared memory in the operating system in favor of message passing between cores that are dedicated to specific functionality [2, 4, 18, 27, 40]. Such designs allow scaling to many-core systems; however, their new designs cannot be easily adopted in legacy systems because they require the re-architecting of major kernel components and probably are best suited for new operating systems. So although such research is crucial to the OS community, our approaches for incremental scaling are also required in practice. A recent study [5] investigated the scalability of the Linux operating system and found that traditional OS kernel designs, such as that of Data ONTAP, can scale effectively for near-term core counts, despite the presence of specific scalability bugs in the CPU scheduler [29]. In contrast, Min, et al. [31] analyze the scalability of five production file systems and find many bottlenecks, including some that may require core design changes.

Recently, many file system and operating system designs have been offered to improve scalability. NOVA [41] is a log-structured file system designed to exploit non-volatile memories that allows synchronization-free concurrency on different files. Hare [19] implements a scalable file system designed for non-cache-coherent multicore processors. The work most similar to our own mitigates contention for shared data structures by running multiple OS instances within virtual machines [7, 36]. In a similar way, MultiLanes [23] and SpanFS [24] create independent virtualized storage devices to eliminate contention for shared resources in the storage stack. Other approaches to OS scaling on multicore systems include reducing OS overhead by collectively managing “many-core” processes [25], tuning the scheduler to optimize use of on-chip memory [6], and even exposing vector interfaces within the OS to more efficiently use parallel hardware [39].

One obvious alternative to data partitioning is the use of fine-grained synchronization, to which many optimizations have been applied. Read-copy update is an approach to improve the performance of shared access to data structures, in particular within the Linux kernel [30]. Both flat combining [16] and remote core locking [28] improve the efficiency of synchronization by assigning particular threads the role of executing all critical sections for a given lock. Specifically in the context of hierarchical data structures, intention locks [15] synchronize access to one branch of a hierarchy, and DomLock [22] makes such locking more efficient. Our approach provides lock-free access to hierarchical structures in the common case, although the locking introduced by Hybrid Waffinity certainly stands to benefit from some of these optimizations. Parallel execution can also be provided via runtime systems that infer task data-independence without explicit data partitions [3, 34].

The database community has long used data partitioning to facilitate parallel and distributed processing of transactions. Many algorithms exist for deploying a scalable database partitioning [1]. The process of defining partitions for optimal performance can also be done on the fly while monitoring workload patterns [20]. The Dora [32] and H-Store [37] models provide data partitions such that operations in a partition can be performed without requiring fine-grained locking. Other work [21, 38] seeks to address the problem of accesses to multiple partitions in a partitioned database. In these proposals, operations with a partition are serialized (and therefore lock-free), but transactions applied to multiple partitions are facilitated through use of a two-phase commit protocol (or similar). In our Hierarchical Waffinity model, we instead superimpose a locking model on top of the partitioned data such that the operation can run safely from within a single partition.

Hierarchical data partitioning has also been explored. In most cases, the partitioning model is optimized around the presence of a hierarchical computational substrate [8, 9, 10, 12]. In contrast, our work focuses on the hierarchy inherent in the data itself in order to provide ample lock-free parallelism on traditional multicore systems. Similar work has been done to optimize scientific applications by using a hierarchical partitioning of the input data [14].

8 Conclusion

In this paper, we have presented the evolution of the multiprocessor model in WAFL, a high-performance production file system. At each step along the way we have allowed continued multiprocessor scaling without requiring significant changes to the massive and complicated code base. Through this work we have 1) provided a simple data partitioning model to parallelize the majority of file system operations; 2) removed excessive serialization constraints imposed by Classical Waffinity on certain workloads by using hierarchical data partitioning; and 3) implemented a hybrid model based on the targeted use of fine-grained locking within a larger data-partitioned architecture. Our work has resulted in substantial scalability and performance improvements on a variety of critical workloads, while meeting aggressive product release deadlines, and it offers an avenue for continued scaling in the future. We also believe that the techniques discussed in this paper can influence other systems, because the hierarchical model is relevant to any hierarchically structured system and the hybrid model can be employed in insufficiently scalable systems based on partitioning.

References

[1] Sanjay Agrawal, Vivek Narasayya, and Beverly Yang. Integrating vertical and horizontal partitioning into automated physical database design. In *Proceedings of the Internal Conference on Management of Data (SIGMOD)*, 2004.

[2] Andrew Baumann, Paul Barham, Pierre-Evariste Dagand, Tim Harris, Rebecca Isaacs, Simon Peter, Timothy Roscoe, Adrian Schupbach, and Akhilesh Signhania. The multikernel: A new OS architecture for scalable multicore systems. In *Proceedings of the Symposium on Operating System Principles (SOSP)*, 2009.

[3] Micah J. Best, Share Mottishaw, Craig Mustard, Mark Roth, Alexandra Federova, and Andrew Brownsword. Synchronization via scheduling. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 2011.

[4] Silas Boyd-Wickizer, Haibo Chen, Rong Chen, Yandong Mao, M. Frans Kaashoek, Robert Morris, Aleksey Pesterev, Lex Stein, Ming Wu, Yuehua Dai, Yang Zhang, and Zheng Zhang. Corey: An operating system for many cores. In *Proceedings of the Symposium on Operating System Design and Implementation (OSDI)*, 2008.

[5] Silas Boyd-Wickizer, Austin T. Clemens, Yandong Mao, Aleksey Pesterev, M. Frans Kaashoek, Robert Morris, and Nikolai Zeldovich. An analysis of linux scalability to many cores. In *Proceedings of the Symposium on Operating System Design and Implementation (OSDI)*, 2010.

[6] Silas Boyd-Wickizer, Robert Morris, and M. Frans Kaashoek. Reinventing scheduling for multicore systems. In *Proceedings of the Workshop on Hot Topics in Operating Systems (HotOS)*, 2012.

[7] Edouard Bugnion, Scott Devine, Kinshuk Govil, and Mendel Rosenblum. Disco: Running commodity operating systems on scalable multiprocessors. *ACM Transaction on Computer Systems*, 15(4), 1997.

[8] M. Chu, K. Fan, and S. Mahlke. Region-based hierarchical operation partitioning for multicluster processors. In *ACM SIGPLAN Notices*, 2003.

[9] M. Chu, R. Ravindra, and S. Mahlke. Data access partitioning for fine-grain parallelism on multicore architectures. In *Proceedings of the International Symposium on Microarchitecture (MICRO)*, 2007.

[10] D. Clarke, A. Ilic, A. Lastovetsky, and L. Sousa. Hierarchical partitioning algorithm for scientific computing on highly heterogeneous CPU+GPU clusters. In *Proceedings of the European Conference on Parallel and Distributed Computing (Euro-Par)*, 2012.

[11] Peter Denz, Matthew Curtis-Maury, and Vinay Devadas. Think global, act local: A buffer cache design for global ordering and parallel processing in the WAFL file system. In *Proceedings of the Internal Conference on Parallel Processing (ICPP)*, 2016.

[12] H. Dutta, F. Hannig, and J. Teich. Hierarchical partitioning for piecewise linear algorithms. In *Parallel Computing in Electrical Engineering*, 2006.

[13] John K. Edwards, Daniel Ellard, Craig Everhart, Robert Fair, Eric Hamilton, Andy Kahn, Arkady Kanevsky, James Lentini, Ashish Prakash, Keith A. Smith, and Edward Zayas. FlexVol: flexible, efficient file volume virtualization in WAFL. In *USENIX Annual Technical Conference (ATC)*,

2008.

[14] O. Ergul and L. Gurel. A hierarchical partitioning strategy for an efficient parallelization of the multilevel fast multipole algorithm. In *IEEE Transactions on the and Propagation*, 2009.

[15] J. N. Gray, R. A. Lorie, and G. R. Putzolu. Granularity of locks in a shared data base. In *Proceedings of the International Conference on Very Large Data Bases (VLDB)*, 1975.

[16] Danny Hendler, Itai Incze, Nir Shavit, and Moran Tzafrir. Flat combining and the synchronization-parallelism tradeoff. In *Proceedings of the Symposium on Parallelism in Algorithms and Architectures (SPAA)*, 2010.

[17] Dave Hitz, James Lau, and Michael Malcolm. File system design for an NFS file server appliance. In *USENIX Winter Technical Conference*, 1994.

[18] David A. Holland and Margo I. Seltzer. Multicore OSes: Looking forward from 1991, er, 2011. In *Proceedings of the Workshop on Hot Topics in Operating Systems (HotOS)*, 2011.

[19] Charles Gruenwald III, Filippo Sironi, M. Frans Kaashoek, and Nickolai Zeldovich. Hare: a file system for non-cache-coherent multicores. In *Proceedings of the European Conference on Computer Systems (EuroSys)*, 2015.

[20] A. Jindal and J. Dittrich. Relax and let the database do the partitioning online. In *Enabling Real-Time Business Intelligence*, 2012.

[21] Evan P. C. Jones, Daniel J. Abadi, and Sameul Madden. Low overhead concurrency control for partitioned main memory databases. In *Proceedings of the Internal Conference on Management of Data (SIGMOD)*, 2010.

[22] Saurabh Kalikar and Rupesh Nasre. DomLock: A new multi-granularity locking technique for hierarchies. In *Proceedings of the Symposium on Principles and Practices of Parallel Programming (PPoPP)*, 2016.

[23] Junbin Kang, Benlong Zhang, Tianyu Wo, Chunming Hu, and Jinpeng Huai. MultiLanes: Providing virtualized storage for OS-level virtualization on many cores. In *Proceedings of Conference on File and Storage Technologies (FAST)*, 2014.

[24] Junbin Kang, Benlong Zhang, Tianyu Wo, Weiren Yun, Lian Du, Shuai Ma, and Jinpeng Huai. SpanFS: A scalable file system on fast storage devices. In *Proceedings of the USENIX Annual Technical Conference (ATC)*, 2015.

[25] Kevin Klues, Barret Rhoden, Andrew Waterman, David Zhu, and Eric Brewer. Processes and resource management in a scalable many-core OS. In *Proceedings of the Workshop on Hot Topics in Parallelism (HotPar)*, 2010.

[26] Charles E. Leiserson. The Cilk++ concurrency platform. In *Proceedings of the Design Automation Conference (DAC)*, 2009.

[27] Min Li, Sudharshan S. Vazhkudai, Ali R. Butt, Fei Meng, Xiaosong Ma, Youngjae Kim, Christian Engelmann, and Galen Shipman. Functional partitioning to optimize end-to-end performance on many-core architectures. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis (SC)*, 2010.

[28] Jean-Pierre Lozi, Florian David, Gael Thomas, Julia Lawall, and Gilles Muller. Remote core locking: Migrating critical-section execution to improve the performance of multithreaded applications. In *Proceedings of the USENIX Annual Technical Conference (ATC)*, 2012.

[29] Jean-Pierre Lozi, Baptiste Lepers, Justin Funston, Fabien Gaud, Vivien Quema, and Alexandra Federova. The Linux scheduler: a decade of wasted cores. In *Proceedings of the European Conference on Computer Systems (EuroSys)*, 2016.

[30] Paul E. McKenney, Dipankar Sarma, Andrea Arcangeli, Andi Kleen, Orran Krieger, and Rusty Russell. Read copy update. In *Proceedings of the Ottawa Linux Symposium*, 2002.

[31] Changwoo Min, Sanidhya Kashyap, Steffen Maass, Woonhak Kang, and Taesoo Kim. Understanding manycore scalability of file systems. In *Proceedings of the USENIX Annual Technical Conference (ATC)*, 2016.

[32] I. Pandis, R. Johnson, N. Hardavellas, and A. Ailamaki. Data-oriented transaction execution. In *Proceedings of the VLDB Endowment (PVLDB)*, 2010.

[33] Hugo Patterson, Stephen Manley, Mike Federwisch, Dave Hitz, Stever Kleiman, and Shane Owara. SnapMirror: File system based asynchronous mirroring for disaster recovery. In *Proceedings of Conference on File and Storage Technologies (FAST)*, 2002.

[34] Martin C. Rinard and Monica S. Lam. The design, implementation, and evaluation of Jade. *ACM Transaction on Programming Languages and Systems*, 20(1), 1998.

[35] SPEC SFS (System File Server) benchmark. `www.spec.org/sfs2008`. 2014.

[36] Xiang Song, Haibo Chen, Rong Chen, Yuanxuan Wang, and Binyu Zang. A case for scaling applications to many-core with OS clustering. In *Proceedings of the European Conference on Computer Systems (EuroSys)*, 2011.

[37] 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 International Conference on Very Large Data Bases (VLDB)*, 2007.

[38] Alexander Thomson, Thaddeus Diamond, Shu-Chun Weng, Kun Ren, Philip Shao, and Daniel J. Abadi. Calvin: fast distributed transactions for partitioned database systems. In *Proceedings of the Internal Conference on Management of Data (SIGMOD)*, 2012.

[39] Vijay Vasudevan, David G. Andersen, and Michael Kaminsky. The case for VOS: The vector operating system. In *Proceedings of the Workshop on Hot Topics in Operating Systems (HotOS)*, 2011.

[40] David Wentzlaff and Anant Agarwal. Factored operating systems (fos): The case for a scalable operating system for multicores. *Operating Systems Review*, 43(2), 2009.

[41] Jian Xu and Steven Swanson. NOVA: A log-structured file system for hybrid volatile/non-volatile main memories. In *Proceedings of Conference on File and Storage Technologies (FAST)*, 2016.

Copyright notice

CLARINET: WAN-Aware Optimization for Analytics Queries

Raajay Viswanathan° *Ganesh Ananthanarayanan*† *Aditya Akella*°
°*University of Wisconsin-Madison* †*Microsoft*

Abstract

Recent work has made the case for geo-distributed analytics, where data collected and stored at multiple datacenters and edge sites world-wide is analyzed *in situ* to drive operational and management decisions. A key issue in such systems is ensuring low response times for analytics queries issued against geo-distributed data. A central determinant of response time is the query execution plan (QEP). Current query optimizers do not consider the network when deriving QEPs, which is a key drawback as the geo-distributed sites are connected via WAN links with heterogeneous and modest bandwidths, unlike intra-datacenter networks. We propose CLARINET, a novel WAN-aware query optimizer. Deriving a WAN-aware QEP requires working jointly with the execution layer of analytics frameworks that places tasks to sites and performs scheduling. We design efficient heuristic solutions in CLARINET to make such a joint decision on the QEP. Our experiments with a real prototype deployed across EC2 datacenters, and large-scale simulations using production workloads show that CLARINET improves query response times by $\geq$ 50% compared to state-of-the-art WAN-aware task placement and scheduling.

1 Introduction

Large organizations, such as Microsoft, Facebook, and Google each operate many 10s-100s of datacenters (DCs) and edge clusters worldwide [1, 5, 6, 13] where crucial services (e.g., chat/voice, social networking, and cloud-based storage) are hosted to provide low-latency access to (nearby) users. These sites routinely gather service data (e.g., end-user session logs) as well as server monitoring logs. Analyzing this *geo-distributed* data is important toward driving key operations and management tasks. Example analyses include querying server logs to maintain system health dashboards, querying session logs to aid server selection for video applications [15], and correlating network/server logs to detect attacks.

Recent work has shown that centrally aggregating and analyzing this data using frameworks such as Spark [48] can be slow, i.e., it cannot support the timeliness requirements of the applications above [24], and can cause wasteful use of the expensive wide-area network (WAN) bandwidth [35, 43, 36]. In contrast, executing the analytics queries *geo-distributedly* on the data stored *in-place* at the sites—an approach called geo-distributed analytics (GDA)—can result in faster query completions [35, 43].

GDA entails bringing *WAN-awareness* to data analytics frameworks. Prior work on GDA has shown how to make query execution (specifically, data and task placement) WAN-aware [43, 35, 36]. This paper makes a strong case for pushing WAN-awareness up the data analytics stack, into *query optimization*. While it can substantially lower GDA query completion times, it requires radical new approaches to query optimization, and rethinking the division of functionalities between query optimization and execution.

The query optimizer (QO) takes users' input query/script and determines an optimal *query execution plan* (QEP) from among many equivalent QEPs that differ in, e.g., their ordering of joins in the query. QOs in modern analytics frameworks [2, 7], largely use database technology developed over 30+ years of research. These QOs consider many factors (e.g., buffer cache and distribution of column values) but largely ignore the network because they were designed for a single-server setup. Some parallel databases considered the network, but they model the cost of *any* over-the-network access via a single parameter. This is less problematic within a DC where the network is high-bandwidth and homogeneous. Geo-distributed clusters, on the other hand, are connected by WAN links whose bandwidths are *heterogeneous* and *limited* (§2.1), varying by over 20×, because of differences in provisioning of WAN links as well as usage by different (non-analytics) applications.

Given this heterogeneity, existing network-agnostic QOs can produce query plans that are far from optimal (§2.2). For example, QOs decide the ordering of multi-way joins purely based on the *size* of the intermediate outputs. However, this can lead to heavy data transfer over thin WAN links, thereby inflating completion times. Likewise, today's QOs optimize one query at a time; as such, when multiple queries are issued simultaneously, their individual QEPs can contend for the same WAN links. Thus, we need a new approach for WAN-aware multi-query optimization.

Arguably, because QOs are upper-most in analytics stacks, them being network-agnostic fundamentally limits the benefits from downstream advances in task placement/scheduling [21, 43, 35, 36]. However, as data analytics queries are DAGs of interconnected tasks, WAN-aware query planning itself has to be performed *in concert* with placement and scheduling of the queries' tasks and intermediate network transfers (§2.2), in contrast with most existing systems where these are conducted in isolation. This is because task placement impacts which WAN links are exercised by a given QEP, and scheduling impacts when they are exercised, both of which determine if the QEP is WAN-optimal. Unfortunately, formulating an optimal solution for such *multi-query network-aware joint query planning, placement, and scheduling* is computationally intractable.

We develop a novel heuristic for the above problem. First, we show how to compute the WAN-optimal QEP for a single query, which includes task placement and scheduling (§4). For tractability, our solution relies on reserving WAN links for scheduled (but yet to execute) tasks/transfers; however, we show that such link reservations lead to faster query completions in practice.

Given a batch of n queries, we order them based on their individually optimal QEPs' expected completion time; the QEP for the i^{th} query is chosen considering the WAN impact of the preceding $i-1$ queries. This mimics shortest-job first (SJF) order while allowing for cross-query optimization (§5.1). However, it results in bandwidth fragmentation (due to task dependencies), thereby hurting completion times. To overcome this, our final heuristic considers groups of $k \leq n$ queries from the above order and explores how to compact their schedules tightly in time, while obeying inter-task ordering (§5.2). The result is a cross-query schedule that veers from SJF but is closer to work-conserving, and offers low average completion times for GDA queries. We also extend the heuristic to accommodate fair treatment of queries, minimizing WAN bandwidth costs, and online query arrivals (§5.3).

We have built our solution into CLARINET, a WAN-aware QO for Hive [3]. Instead of introducing WAN-awareness inside existing QOs [2, 7], CLARINET is architecturally *outside* of them. We modify existing QOs to simply output all the functionally equivalent QEPs for a query, and CLARINET picks the best WAN-aware QEP per query, as well as task placement and scheduling which it provides as *hints* to the query execution layer. Our design allows any analytics system to take advantage of CLARINET with minimal changes.

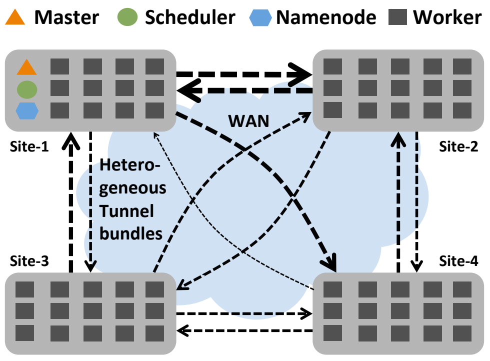


Figure 1: Architecture of GDA Systems

We deploy a CLARINET prototype across 10 regions on Amazon EC2, and evaluate it using realistic TPC-DS queries. We also conduct large scale trace-driven simulations using production workloads based on two large online service providers. Our evaluation shows that, compared to the baseline that uses network-agnostic QO and task placement, CLARINET can improve the average query performance by 60-80% percent in different settings. We find that CLARINET's joint query planning and task placement/scheduling doubles the benefits compared to state-of-the-art WAN-aware placement/scheduling.

2 Background and Motivation

In this section, we first discuss the architectural details of GDA, focusing on WAN constraints. We then analyze how queries are handled in existing GDA systems.

2.1 Geo-Distributed Analytics

GDA Architecture: In GDA, there is a central *master* at one of the DCs/edge sites where queries—written, e.g., in SparkSQL [7], HiveQL [3], or Pig Latin [33]—are submitted. For every query, the QO at the master constructs an *optimized query execution plan* (QEP), essentially, a DAG of many interdependent *tasks*. A centralized scheduler places tasks in a QEP at nodes across different sites based on resource availability and schedules them based on task dependencies. [1]

[1] Typically, the task scheduler, the namenode of the distributed file system, and the master all run at the same site to reduce inter-process communication latencies between them. However, it is possible to distribute them across different processing sites.

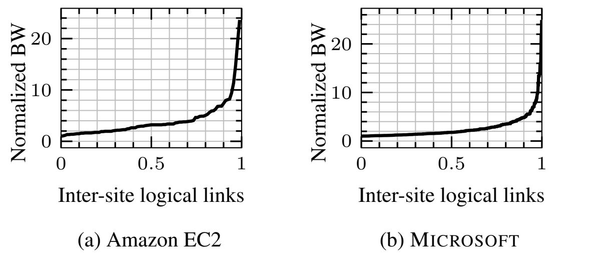


Figure 2: Distribution of bandwidth between data processing sites for Amazon EC2 and a large OSP. The bandwidths reported are normalized with respect to the minimum observed. For Amazon EC2, the bandwidth between a pair of sites is obtained through active measurements using iperf. The minimum bandwidth obtained over 10-minute interval is taken as the guaranteed bandwidth. For MICROSOFT, we use the topology and traffic information from applications to compute the guaranteed bandwidth.

WAN Constraints: The sites are inter-connected by a WAN which we assume is optimized using MPLS-based [45] or software-defined traffic engineering [23, 20, 26]. In either case, end-to-end tunnels are established by the WAN manager for forwarding analytics traffic between all site-pairs. The WAN manager updates tunnel capacities in response to background traffic shifts. The running time of queries is typically lower than the interval between WAN configuration changes ($\sim$ 10 - 15 minutes [20, 23]); thus, we assume that the bandwidth between site-pairs remains constant for the duration of queries' execution. Thus, we can abstract the WAN as a *logical full mesh* with fixed bandwidth links (fig. 1).

However, available bandwidth between pairs of sites can differ significantly because of differences in physical topology and traffic matrix of non-analytics applications. Figure 2 highlights this variation between pairs of the 10 Amazon EC2 regions, and for the DCs operated by MICROSOFT. The ratio of the highest to lowest bandwidth is > 20. Also, the bandwidth is 1-2 orders of magnitude less than intra-DC bandwidth (e.g., the maximum inter-site bandwidth is 450Mbps between EC2 regions, where as intra-site bandwidth is 10Gbps). Thus, WAN bandwidth is highly constraining and a significant bottleneck for GDA, in contrast with intra-DC analytics.

2.2 Illustrative Examples for Drawbacks of Current GDA Query Processing

Given a query, its relational operators (e.g., SELECT, GROUPBY, JOIN, TABLESCAN, etc.) are transformed to individual "map" or "reduce" stages in the QEPs compiled by the QO. For example, SELECT and TABLESCAN are transformed to a map stage, whereas JOIN or GROUPBY are transformed to a reduce stage. If the input tables for these operators are partitioned and/or spread out across different sites, then the execution of downstream reduce stages (for JOIN or GROUPBY) will involve flow of data across the constrained WAN, limiting overall query performance.

Notation	Size
$\lvert\sigma_{\mathrm{A}}(*)\rvert, \lvert\sigma_{\mathrm{X}}(*)\rvert, \lvert\sigma_{Y\lvert \mathrm{Z}}(*)\rvert$	200 GB, 160GB, 25 GB
$\lvert\sigma_{\mathrm{A}\lvert\mathrm{X}}(\mathrm{WS}) \bowtie \sigma_{\mathrm{A}\lvert\mathrm{X}}(\mathrm{SS})\rvert$	12 GB
$\lvert\sigma_{\mathrm{A}\lvert\mathrm{X}}(\mathrm{SS}) \bowtie \sigma_{\mathrm{A}\lvert\mathrm{X}}(\mathrm{CS})\rvert$	10 GB
$\lvert\sigma_{\mathrm{A}\lvert\mathrm{X}}(\mathrm{WS}) \bowtie \sigma_{\mathrm{A}\lvert\mathrm{X}}(\mathrm{CS})\rvert$	16 GB

Table 1: Selectivity and join cardinality estimates

In what follows, we argue that because existing QOs do not account for such WAN constraints, their chosen QEP for a query may be sub-optimal. Because multiple queries can contend simultaneously for limited WAN bandwidth, QOs for GDA must account for two additional issues: (a) consider task placement and network transfer scheduling when determining a QEP's quality, and (b) plan for multiple queries at once.

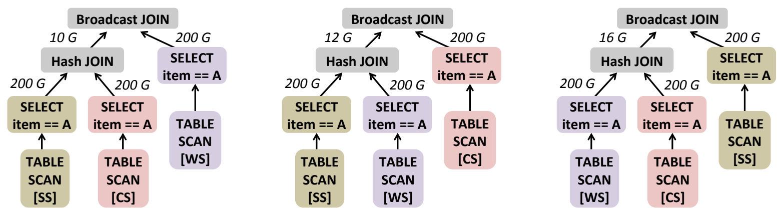
Modern QOs employ a combination of heuristic techniques as well as cost based optimization (CBO) to determine the best execution plan for each query. Heuristic optimization leverages widely accepted techniques — e.g., predicate push down and partition pruning — for reducing query execution times.

CBO explores the space of all possible QEPs — e.g., those generated by considering alternate join ordering of tables — and chooses the one with the least cost. The cost of a QEP is based on a *cost model*, which captures the cost of accessing a single byte of data over a resource, and cardinality estimates of intermediate data based on individual and cross-table statistics, histograms of column values, etc. CBOs also account for a variety of factors, including availability of buffer cache and indexes. State-of-the-art technologies for accurate cardinality estimation and cost modeling have been developed over 30+ years of research. However, most have focused on single server systems which ignore the network as a factor in determining query performance [2, 10, 17]. Even parallel databases model the network as a "single pipe" which essentially assumes the entire network is *homogeneous* [34, 14, 25, 12, 32, 41, 38, 46, 49, 47, 28]. Such simple models are clearly insufficient to account for heterogeneous WAN bandwidths. Yet, this is clearly important in GDA (as shown in §2.1).

Importance of network-aware query optimization: Consider a three-way join, $\mathrm{Q_A}$: $\sigma_{\mathrm{A}}(\mathrm{WS}) \bowtie \sigma_{\mathrm{A}}(\mathrm{SS}) \bowtie \sigma_{\mathrm{A}}(\mathrm{CS})$ shown in fig. 3(a), that compares overall sales for a set of items (starting with 'A') across three different tables; the tables are spread across different sites inter-connected by a WAN (fig. 4). Q_{A} can be executed through three different QEPs (figs. 3(b)–3(d)); one each from three different join orders. Table 1 lists the sizes of different intermediate outputs.

```
SELECT
  SS.item as item,
  SUM(SS.sales),
  SUM(WS.sales),
  SUM(CS.sales)
FROM store_sales SS,
  web_sales WS,
  cat_sales CS
WHERE SS.item == CS.item
  AND SS.item == WS.item
GROUP BY item
HAVING item STARTSWITH 'A'
```

(a) A sample SQL Query



(b) QEP-1 (c) QEP-2 (d) QEP-3

Figure 3: An example SQL query and its different query execution plans. Each QEP corresponds to a different join order. Note how the selectivity predicate is pushed down to minimize the records processed during the joins.

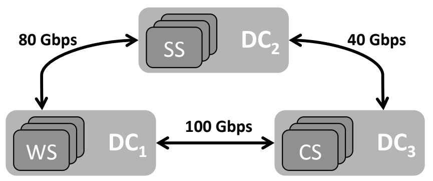


Figure 4: Three sites that are interconnected by bidirectional WAN links. Each site contains a unique table.

A network agnostic QO, or one that models the entire network by a single parameter, will pick QEP-1 since it has the least output cardinality after the first join. QEP-1 will take 20.5s: the first join, $(\sigma_A(SS) \bowtie \sigma_A(CS))$, will be implemented as a *hash join* since it involves large tables; by placing reducers uniformly across sites DC_2 & DC_3 the join involves 100GB of data transfer in either direction. Over a 40Gbps link, this transfer takes 20s. The second join will be implemented as a broadcast join since one of the tables is small. It involves transferring, 10GB of data spread across sites DC_2 & DC_3 to DC_1. The bottleneck is the transfer on the 80Gbps link which takes 0.5s.

Contrast this with QEP-3 that joins tables WS and CS first. Even though it has the highest cardinality for intermediate data, it might be advantageous because sites DC_1 & DC_3 have high bandwidth between them. By placing tasks uniformly between sites DC_1 & DC_3, the first join would take only 8s (100GB over 100Gbps link). The second join takes 1.6s (8GB over 40Gbps link). QEP-3 completes in 9.6s, or ≈53% faster.

Importance of considering placement in QEP selection: For each QEP, the exact pattern of traffic between the data processing sites is dependent not only on the nature of data flow between stages (e.g, scatter-gather, one-to-one) but also on the placement of tasks in each stage. Thus, placement must be taken into account in assessing QEP quality. Placement matters due to contention from currently running GDA queries.

Consider a scenario where an already running query is using the logical links, $DC_1 \rightarrow DC_3$ & $DC_2 \rightarrow DC_1$. Without control over task placement, a network-aware optimizer would choose QEP-1 for Q_A to avoid links already being utilized. Thus, its running time would be 20.5s. However, by choosing QEP-3 and placing all reduce tasks for the first join at DC_1, we can completely avoid $DC_1 \rightarrow DC_3$ & $DC_2 \rightarrow DC_1$ and finish in 17.6s.[2]

Importance of considering scheduling in QEP selection: Similar to placement, the impact of scheduling of network transfers should also be accounted for in assessing a QEP. Consider a query Q_X that is similar to Q_A in structure, but operates on a different slice of data, say, items starting with 'X'. Say Q_X arrives soon after two simple two-way joins, $Q_Y : \sigma_Z(WS) \bowtie \sigma_Z(CS)$ and $Q_Z : \sigma_Z(WS) \bowtie \sigma_Z(SS)$, start to execute. The selectivity information for input datasets of the queries is shown in Table 1. Being two-way joins, Q_Y and Q_Z have no choice of QEPs; they utilize the WAN bandwidth between DC_1 and DC_3 & DC_1 and DC_2, respectively. The joins take 5s each.

Without control over scheduling, QEP-1 is the best choice for executing Q_X (since it avoids the links used by Q_Y and Q_Z). Its completion time is 16.5s.[3] However, if we can control scheduling of queries, we can still choose QEP-3 for Q_X and delay its network transfers by 5s. The completion time is lowered to 13s.[4]

Multi-query optimization: QOs in modern stacks, e.g., Hive and SparkSQL, optimize each query individually. Resource contention among concurrent queries is potentially left to be resolved at the execution layer through scheduling and task placements. However, under scenarios where contention cannot be resolved, *jointly determining the QEP* for all queries provides better opportunities to avoid resource contention thereby resulting in lower query completion times. Classic multi-query database QOs [40] leverage efficient reuse of common sub-expressions across queries [39], shared scans [44] and sharing of buffer caches [19], but they do not model the network similar to single-query QOs.

[2] WS ⋈ CS takes 16s to transfer 200GB over 100Gbps, and the next join takes 1.6s to send 16GB over 80Gbps link.

[3] The first join of QEP-1 utilizes the bandwidth between DC_2 and DC_3 and takes 16s. The bottleneck for the second join–which starts after Q_Z completes—is the 80Gbps link and transferring 5Gb over it takes 0.5s.

[4] The first join of QEP-3 takes 6.4s (80GB between DC_1 and DC_3). The second join takes 1.6s to transfer 8GB from DC_3 to DC_2. With a delay of 5s (waiting for two-way join to finish) completion time is 13s.

Consider a case where Q_A and Q_X arrive concurrently. A network-aware query optimizer will choose the same QEP for both the queries resulting in contention for bandwidth on links between DC_1 & DC_3. The scheduler, to optimize for average completion time, will execute the shortest query first (Q_X in this case); the average running time will then be 12s. However, by choosing QEP-2 and QEP-3 for queries Q_X and Q_A respectively, we can completely avoid link contention and keep the average completion time at 9.4s (9.2 for Q_X and 9.6s for Q_A).

3 CLARINET's Design

Accomplishing multi-query network-aware plan selection and task placement/scheduling requires an analytics framework where a single entity is simultaneously responsible for both QEP selection and scheduling. Current big-data analytics stacks, however are highly modular with individual components being developed and operated independently. Realizing joint optimization in such a setting would require radical changes.

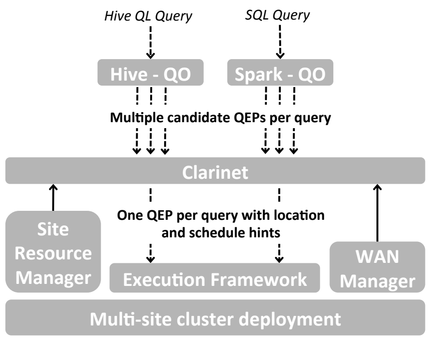

Hive QL Query
SQL Query
Hive - QO
Spark - QO
Multiple candidate QEPs per query
Clarinet
One QEP per query with location and schedule hints
Site Resource Manager
Execution Framework
WAN Manager
Multi-site cluster deployment

Figure 5: CLARINET's late-binding design

CLARINET's design (Figure 5) addresses this challenge via *late-binding*. In CLARINET, the QOs are modified to provide a set of *functionally equivalent* QEPs (QEP-Set)[5] to an intermediate *shim* layer. The shim layer (CLARINET) collects the QEP-Sets from multiple queries/QOs and computes a single, optimal QEP per query as well as location and scheduling *hints*, that it forwards to the execution framework.

Each node (operator) in a QEP, forwarded from QOs to CLARINET, is annotated with its output cardinality and parallelism as estimated by the QO. The cardinality represents the total amount of data transferred from the current operator to its successor operator. As this is data that will potentially be sent over the WAN, cardinality directly affects QEP selection. The operator parallelism decides the number of tasks to be spawned for each operator; the location and scheduling hints suggested by CLARINET correspond to location (at data center level) and start time for each task.

[5] A dynamic programming based QO will generate exponentially many query plans for each query. We limit the size of the QEP-Set by placing a bound (5 seconds) on time spent in exploring multiple query plans.

The late binding approach offers several **advantages**. First, given the complexity of QOs, modifying their internal cost model to account for the WAN is quite challenging. Also, QOs with widely different objectives (Calcite [2] vs Catalyst [10]) have to be modified individually. E.g., SparkSQL's QO [2] should factor availability of in-memory caches of RDDs [48] against WAN costs. By design, CLARINET requires no changes to a QO's internal cost model. Any QO that can provide multiple QEPs based on its current cost model can be made WAN-aware through CLARINET. Second, by introducing an intermediate layer, CLARINET alleviates (*i*) the analytics application (e.g., Hive) from making scheduling decisions and joint query optimization, (*ii*) the execution layer from making any network specific scheduling/placement decisions. This minimizes code changes to both the application and execution frameworks. Third, WAN awareness does not come at the cost of the existing query optimizations. An application can avoid the WAN from interfering with plan selection by exposing only its QO's chosen best QEP.

Problem statement, and assumptions: Given a set 'n' queries, CLARINET receives QEP-Sets, $QS_j, j \in [1, ..., n]$ from the QOs corresponding to each query. Among the exponentially many combinations, the objective is to select exactly one QEP from each QS_j along with task locations and schedules, such that the average query run time is minimized. Here, task locations determine the site at which tasks are executed, whereas the schedule determines the start times of each task.

For analytical tractability, we require that the tasks are scheduled such that network transfers on logical links between sites *do not temporally overlap* with one another. This allows us to accurately determine the duration of network transfers and reduce the QEP selection problem to a well studied job-shop scheduling problem. Such time multiplexing (or non-overlap) also has the advantage that resource sharing can be enforced through scheduling; (weighted) bandwidth sharing on the other hand requires additional per transfer rate-control on top of the rate control already enforced by the WAN manager. Crucially, the non-overlapped assumption does not affect the quality of the solution. This is because, as we prove below, any optimal schedule has an equivalent optimal non-overlapped schedule.

Theorem: A schedule, $\mathcal{S}$, of interdependent transfers over multiple network resources, where each transfer is allocated an arbitrary time-varying share of available network bandwidth on a single resource, can be converted into an equivalent interruptible schedule, $\mathcal{N}$, such that no two transfers in $\mathcal{N}$ share a resource at any given point in time, and the completion time of a transfer in $\mathcal{N}$ is not greater than its completion time in $\mathcal{S}$.

Proof sketch: For a network transfer, f, on a resource, let $s(f)$ and $e(f)$ be the start and end times based on schedule, $\mathcal{S}$. For each resource, the start and end times of all its transfers can be viewed as its release time and deadline respectively. Converting $\mathcal{S}$ to $\mathcal{N}$, can be achieved by determining the *earliest deadline first* (EDF) schedule of flows for each resource independently. Given $s(f)$ and $e(f)$, an EDF schedule is feasible since S is a complete schedule. For a detailed proof, refer [42].

We simplify further and focus on obtaining *non-interruptible* transfer schedules, because, implementing interruptible transfers requires significant changes to query execution. However, such schedules do not permit perfect "packing" of transfers across links. The resulting fragmentation of link capacity delays scheduling of network transfers, and inflates completion times. Essentially, CLARINET incorporates a clever approach—which we develop gradually in the next two sections—that systematically combats such resource fragmentation and optimizes average query completion times.

In sum, our simplifying assumptions do not impact CLARINET's effectiveness. However, even with these assumptions, computing the best cross-query QEPs along with task placements and schedules is NP-hard. In fact, the problem is hard even for a single query [30, 31].

Our approach is as follows: we start with an effective heuristic for the best single-query QEP, that decouples placement and scheduling (§4). We then use this to gradually develop our multi-query heuristic (§5).

4 Single Query WAN-Awareness

At a high level, WAN-aware QEP selection for a single query proceeds as follows: for every QEP in the query's QEP-Set, we determine the placement and schedule of tasks such that its running time is minimized. The QEP with the shortest running time is then selected. Because of inherent hardness of joint placement and scheduling of DAGs [30, 31], CLARINET's approach is to decouple them, as described next.

4.1 Assigning Locations to Tasks in a QEP

Given a QEP, for tasks with no dependencies (e.g., map tasks) we use the common approach of "site-locality", i.e., their locations are the same as the location of their input data. The placement of intermediate (reduce) tasks is decided based on the amount and location of intermediate data generated by their parents, along with the bandwidths at the sites.

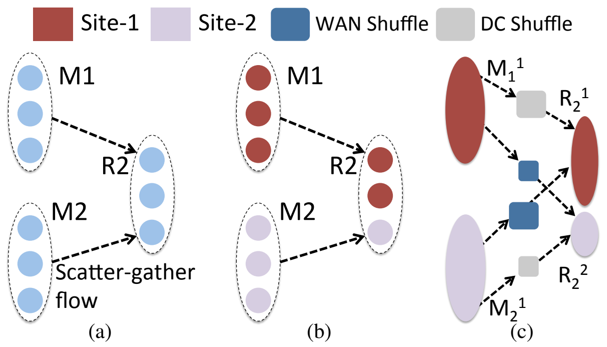


Figure 6: (a) shows a simple MR job with 3 tasks in each stage. (b) shows the same job with placement information (color-coded) for all the tasks. (c) shows the corresponding augmented DAG with tasks in the same stage and location coalesced to one. M_*^* and R_*^* are sub-stages in the augmented DAG. Shuffle tasks representing network transfer between tasks at different locations are shown explicitly.

We decide the task placements for a QEP iteratively for each of its stages in topological order. Since the query plan specifies a partial ordering between dependent stages, stages with no order among them can be simultaneously scheduled. To ensure that the placement decision for these stages takes into account other stages' decisions, we *reserve* a block of time on logical links for network transfers (consistent with our non-overlapped assumption).

Formulation: The optimal task placement for a stage is obtained by solving a linear program which takes as input the following: *(i)* the distribution of output data (D_ℓ) from the predecessor stages across sites (ℓ), *(ii)* the inter-site WAN bandwidths ($B_{\ell_1}^{\ell_2}$), and *(iii)* the length of time ($\tau_{\ell_1}^{\ell_2}$) for which stages (which do not have ordering with respect to current stage) have reserved inter-site links.[6] The best distribution of tasks (r_ℓ) across sites is obtained by solving the following problem:

$$\min_{r} \sum_{\ell_1,\ell_2} \frac{D_{\ell_1} r_{\ell_2}}{B_{\ell_1}^{\ell_2}} + \tau_{\ell_1}^{\ell_2} \tag{1a}$$

$$\text{such that} \quad r_{\ell_2} \geq 0 \tag{1b}$$

$$\sum_{\ell_2} r_{\ell_2} = 1 \tag{1c}$$

Once the locations of the reducers are fixed, we use the resulting traffic pattern to update the durations for which resources are blocked for later stages. E.g., between ℓ_1 and ℓ_2 we increment the duration, $\tau_{\ell_1}^{\ell_2}$, by $\frac{D_{\ell_1} r_{\ell_2}}{b_{\ell_1}^{\ell_2}}$.

4.2 Scheduling tasks in a QEP

In contrast to scheduling within a DC, scheduling a QEP in a geo-distributed setting involves scheduling both the compute phase of a task and the transfer

[6] ℓ, ℓ_1, ℓ_2 are indices over the set of data processing sites

of input data from remote sites. To explicitly model these network transfers, we augment the DAG of tasks representing the QEP to include vertices (called *shuffle tasks*) corresponding to network transfers; fig. 6 shows an example. We assume that the compute phase of a task can only start after all its inputs are available at the site. Further, since tasks of a stage that are executed at the same site exercise the same network resource, we coalesce them into a *sub-stage*. This reduces the number of entities that need to be scheduled and the overall scheduling complexity.

We formulate the scheduling as a binary integer linear program that takes as input the following: *(i)* the coalesced DAG augmented with shuffle tasks, henceforth called *augmented-DAG*, which captures dependencies among tasks, and *(ii)* the duration of compute and shuffle tasks. The duration of a compute task is same as the expected running time of the task in a intra-DC setting. The duration of shuffle between sites is estimated as the ratio of data transferred to the WAN bandwidth between the sites. The objective is to determine the optimal start times for all the tasks in the augmented-DAG such that the overall execution time of the QEP is minimized.

Formulation: Let c^i be the augmented-DAG of the i-th QEP in the QEP-Set for the query. Let V^i represent the set of vertices in c^i and $\leq^i$ represent the partial order between them. The start times of the vertices ($s(.)$) should obey the partial order $\leq^i$. Thus, for each pair of ordered vertices, $(u, v) \in \leq^i$, belonging to c^i,

$$s(v) \geq s(u) + d(u) \tag{2}$$

where, $d(.)$ represents the duration of vertices.

We incorporate non-overlapping of flows on network links in our scheduling problem by imposing:

$$s(v) \geq s(u) + d(u) - N(1 - z_{uv}) \tag{3a}$$
$$s(u) \geq s(v) + d(v) - N(z_{uv}) \tag{3b}$$

where u and v are shuffle tasks that contend for the same network link, and z_{uv} indicates v is executed after u; N is large constant. When $z_{uv} = 1$, then Equation 3a ensures that the start time of vertex v is greater than the completion time of vertex u; Equation 3b remains void, since it is satisfied trivially. When $z_{uv} = 0$, the conditions invert. Equations (3a) and (3b) are enforced for all links.

The completion time (Φ^i) of the i-th QEP, is given by:

$$\Phi^i := \max_{u \in V^i} \quad s(u) + d(u) \tag{4}$$

where, u is any vertex in c^i. We solve the program for all the QEPs in the QEP-Set. The one with the smallest duration is chosen to be executed.

Handling currently running queries: We "reserve" network links for tasks already placed and scheduled. Therefore, while computing the best schedule for a QEP, we have to factor in currently running queries that block resources. We add constraints to the above formulation in order to accommodate these currently running queries.

Let $B(r)$ be a set of time intervals for which existing queries block resource r. Let $low(b)$ and $high(b)$ represent the lower and upper bound of an interval $b \in B(r)$. For every vertex u using a network link, we include these two constraints:

$$s(u) \geq high(b) - N(1 - x_{ub}) \tag{5a}$$
$$low(b) \geq s(u) + d(u) - N(x_{ub}) \tag{5b}$$

where x_{ub} is a binary indicator denoting that u is scheduled after the interval b. Like eqs. (3a) and (3b), these constraints kick in alternatively ensuring the transfers do not overlap with intervals that are reserved.

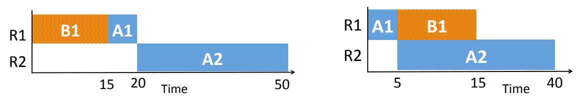
5 Multiple Contending Queries

In this section, we build upon the solution in §4 to solve the problem statement outlined in §3 for a workload of 'n' (>1) queries that arrive simultaneously and compete with each other for the inter-site WAN links.

In §5.1, we first provide a strawman algorithm that emulates shortest-job first (SJF), and iteratively determines the QEP, placement, and schedule for each of the 'n' queries. Unfortunately, the strawman algorithm results in a schedule with link resources being fallow close to 22% of the time (ref. Figure 12(a)) due to resource fragmentation. In §5.2, we present a novel heuristic that builds on the strawman and minimizes resource fragmentation; it combats fragmentation by carefully packing flows from k ($\leq n$) queries at a time from the schedule determined by the strawman. We discuss several enhancements in §5.3.

5.1 Strawman Iterative QEP Selection

Our strawman heuristic is based on shortest-job first (SJF) scheduling. We pick this because it is a well understood scheduling discipline that is typically used to minimize average completion times. Our strawman functions iteratively. In every iteration, we pick the QEP and determine the schedule for exactly one query as follows. For each QEP belonging to the QEP-Set of unscheduled queries, we calculate its duration, placement, and schedule of tasks (using techniques in §4). We then pick the QEP (and thus the query) with *shortest duration* among all the QEPs considered; we do not consider this query for future iterations. At the end of each iteration, we reserve resources required by the QEP chosen. By doing so, we ensure that the running time for the query is not affected by queries considered in future iterations. Further, it ensures that choice of QEPs in future iterations account for the current query's WAN impact, thereby enabling cross-query planning.



(a) SJF schedule (b) Better Schedule

Figure 7: Example highlighting fragmentation of resources with SJF heuristic. Tasks A1 and A2 belong to Job A; A2 is dependent on A1. Job B has only one task, B1. A1 and B1 use resource R1, A2 uses resource R2. In the SJF schedule, resource R2 remains idle until $t = 20$s.

5.2 Final Heuristic to Combat Resource Fragmentation

The above iterative heuristic is not ideal because it can cause links to remain fallow sometimes, *even if* there are other flows which could use those links. See Figure 7; as shown, fragmentation arises because jobs need multiple resources (multiple links in this case) and because of dependencies across tasks. What this shows is that vanilla SJF is not ideal for minimizing average completion times in our setting. If not controlled, underutilization of link resource can delay query completions arbitrarily.

We address this by modifying the above SJF strawman to use a knob (k) that reduces resource fragmentation. The knob allows us to deviate away from the iteratively computed schedule towards a schedule with low fragmentation of resources in a controlled manner. We start with the solution obtained from the iterative algorithm described in §5.1. The solution determines the following: *(i)* $\mathcal{O}$, a total ordering over the queries, based on their time of completion, and *(ii)* the mapping of inter-site flows to network resources (obtained from the choice of QEP and task placement). With this information, our final heuristic creates a constrained schedule as follows:

We maintain a dynamic set, $\mathcal{D}$, consisting of k-shortest queries (based on ordering $\mathcal{O}$), for which at least one task is not yet scheduled. Whenever a flow belonging to a query in $\mathcal{D}$ is available (i.e., at the time when all its predecessors have completed), and the resource it needs is free, we immediately schedule the flow on the resource rather than wait for its start time based on the iterative schedule. If multiple flows meet the criteria, we break ties in favor of short duration flows. When all tasks for a query are scheduled, it is dropped from the dynamic set and a new query is added.

When $k = 1$, only flows belonging to the shortest query can be moved ahead; thus the resulting schedule will be close to the strawman's. When k equals the total number of concurrent queries N, the resulting schedule will have no fallow links (note that the query completion times and the ordering of flows on a resource will be different from that computed using our iterative SJF algorithm). But, it may not offer good performance. This happens because, at high values of k, the initial stages (mappers) of the k QEPs are scheduled first, as they are available immediately. Thus, resources are indiscriminately blocked for later stages for *all* k QEPs, resulting in an increase in average completion times. We evaluate this effect in §7, and show the optimal average completion time benefits of an ideal "sweet-spot value" of k.

Note that in this heuristic, only the schedule is altered; task the placement and the QEP remains the same.

5.3 Enhancements

Fairness: Our heuristic can lead to long queries' start times being pushed significantly to favor shorter running queries. This is not acceptable if the long queries are initiated by different applications that require performance guarantees. To mitigate this bias, we adopt an approach similar to [22]. Essentially we want to ensure that the running time of a query, Q_j, is bounded by $d_j = n \times dur_j$, where n is the number of simultaneously running queries, dur_j is the standalone run time of the query without contention, and d_j denotes the calculated deadline for each query.

Then, we adapt the heuristic in §5.2 as follows. We sort queries in descending order based on a "proximity score"; this score determines how close a query is to its deadline and is obtained as:

$$Proximity_j(t) = 1 - \frac{d_j - t}{d_j} \tag{6}$$

where t is the time at which the dynamic set (§5.2) is updated (upon completion of a query). We pick the top ϵM queries in this sorted order and call them $\mathcal{H}$. Here, ϵ ($0 < \epsilon \leq 1$) is a fairness control knob and M is the number of queries with at least one task not yet scheduled. The dynamic set $\mathcal{D}$ (from §5.2) is then obtained by picking the shortest-k queries from $\mathcal{H}$. If $k > |\mathcal{H}|$, then $\mathcal{D} = \mathcal{H}$. By doing so, we block the tasks of queries that are far from their deadline from being scheduled and prefer those closer to their deadline.

When $\epsilon = 1$, $\mathcal{H}$ contains all the remaining queries and the heuristic is identical to the one in §5.2. When $\epsilon \to 0$, $\mathcal{D}$ contains only queries with highest proximity to fair-share deadlines; thus, offering maximum fairness.

WAN utilization: By favoring QEPs and task placement that result in smaller completion times, CLARINET implicitly reduces the WAN usage. However, unlike recent work [43], CLARINET cannot provide explicit guarantees on WAN usage. To explicitly control WAN usage, we filter from the QEP-Set of all queries those QEPs whose best (in terms of WAN use) task placement results in inter-site WAN usage exceeding a threshold, β. With a limited set of QEPs per query, we then apply techniques in §5.1 and §5.2 for scheduling the transfers.

Online arrivals: We assumed so far that the set of n queries arrive simultaneously. We now extend the heuristic in §5.2 to support online query arrivals. Upon arrival of a new query, we recompute the QEP choice, task placement, and schedule for the current query together with all previous queries for which none of the tasks have started executing. Doing so might alter the QEP and schedule for prior, as yet unexecuted queries based on new information. Changing the QEP for already executing queries would incur wastage of resources; CLARINET does not alter the QEP for those queries.

6 Implementation

We build CLARINET as a stand-alone module that can interface with Hive [3] at the application level and Tez [4] at the execution framework level. We modified Hive and Tez to interface with CLARINET as follows:

Modifications to Hive/Calcite: Hive internally uses the Apache Calcite [2] library as a CBO. Calcite offers two types of QOs: *(i) HepPlanner*, which is a greedy CBO, and *(ii) VolcanoPlanner*, a dynamic programming-based CBO [16], which enumerates all possible QEPs for a query. By default, Hive uses the HepPlanner, but since it does not explore all possible QEPs, we modify Hive to interface with VolcanoPlanner. We further modify VolcanoPlanner to return the operator trees (OPT) representing multiple join orders along with the estimated cardinality (in bytes) for each operator, for each input query. All the OPTs are then compiled to corresponding QEPs by applying heuristic physical layer optimizations like partition pruning, field trimming, etc. The QEPs together constitute the QEP-Set for the query. We find that a typical TPC-DS [8] query has tens of QEPs in its QEP-Set. Each QEP is also annotated with the estimate of intermediate data for each stage; this is used by CLARINET to estimate network transfer times.

Modifications to Tez: CLARINET interfaces with Tez by providing *hints* regarding placement locations and start times for individual tasks. We modify Tez's DAG scheduler to schedule tasks based on these inputs. If a task becomes available before its scheduled start time, we hold it back and schedule it for execution later; a task is never held back beyond its scheduled start time.

Scheduling non-overlapped transfers: CLARINET employs a schedule that requires non-overlap of flows between two sites. Consider the simple MapReduce job similar to one in fig. 6(a). If tasks from two map stages (say, M_1^1 and M_2^1) are executed at the same location, then the transfer of their intermediate data to any downstream task (say, R_1^1) happens in an overlapped fashion; i.e.,

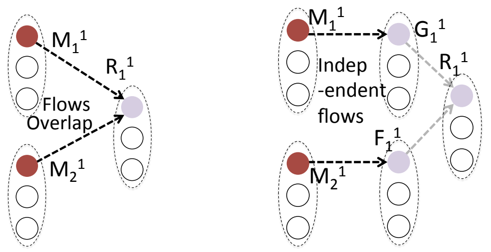


Figure 8: Modification of QEPs forwarded to execution framework by adding relay stages (F and G). Relay stages ensure network transfers fully utilize bandwidth and can be scheduled in a non-overlapped fashion. Here, map tasks M_1^1 and M_2^1 are executed in the same site, whereas reducer task, R_1^1 is executed in a different site. Relay tasks, F_1^1 and G_1^1 are co-located with R_1^1.

when R_1^1 starts executing, it reads data written by both M_1^1 and M_2^1 simultaneously. To enforce non-overlapped transfers by controlling task schedule, we introduce *relay* stages in the QEP (stages F and G in fig. 8(b)). The task in a relay stage does not process data; it reads remote data and writes it locally. Its parallelism and locations are identical to the corresponding reducer stage. By specifying start times of tasks (F_1^1 and G_1^1 in fig. 8(b)) in the relay stage, CLARINET explicitly determines start times of inter-stage shuffles and can ensure they happen in a non-overlapped fashion.

7 Evaluation

We experimentally evaluate CLARINET in realistic settings and against state-of-the-art GDA techniques. We evaluate CLARINET first in a real GDA deployment over 10 Amazon EC2 DCs. We use the standard TPC-DS [8] workload for benchmarking. For evaluating CLARINET at a large scale, we also use traces from analytics queries executed on two OSPs' production clusters. We simulate a GDA setup spread across tens of DCs and executing 1000's of queries. By default, we run CLARINET without the fairness enhancement.

The *de-facto* way in which queries are executed in a Hive-atop-Tez deployment is used as the baseline for comparison. Specifically, query selection and task placement are both network agnostic; here the QEP is selected by Hive's default QO and the reducers are placed uniformly across sites where input data is present. Since we are interested in reducing average completion time, we use our shortest query first heuristic (SJF; §5.1) to schedule the tasks belonging to multiple queries. We call the baseline HIVE+.[7]

Prior work [35] has shown that centrally aggregating raw data to one DC is wasteful. However, they only

[7] The '+' in HIVE+ indicates that the SJF heuristic is used for multiple queries. In a normal deployment,concurrent queries will arbitrarily share WAN bandwidth thereby delaying completion time for all.

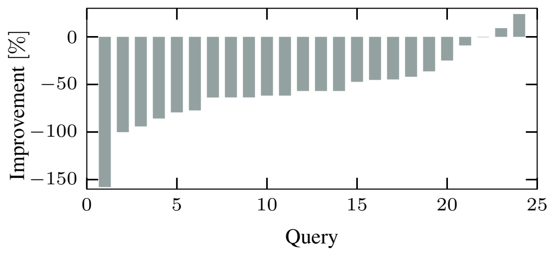


Figure 9: Percentage reduction in running time of HIVESINGLEDC w.r.t. HIVE+ for TPC-DS queries (sorted by increasing gains). The negative gains indicate running times of queries with HIVESINGLEDC is greater than HIVE+.

consider the case where raw input data is centrally aggregated; it is possible to reduce the amount of data sent over the WAN by suitably processing/filtering raw input data. For completeness, we evaluate our baseline (HIVE+) against an alternative that centrally aggregates data after pre-processing; we call this alternative, HIVESINGLEDC. In our implementation, HIVESINGLEDC uses the default QEP chosen by Hive's QO; the map tasks which process (filter) the raw input data are co-located with the data and all reduce tasks are placed in the DC with maximum intermediate data after map stages.

We also study HIVE-IR+ in simulation. HIVE-IR+ uses the QEP chosen by Hive but decisions on placement and scheduling are made using algorithms described in §4 and §5. The IR in HIVE-IR+ stands for Iridium [35], a state-of-the-art scheme for WAN-aware data/task placement. CLARINET's task placement is similar to Iridium's [35]. However, we use the "+" suffix since Iridium only does network-aware data/task placement, whereas HIVE-IR+ also does network-aware transfer scheduling. Comparing CLARINET and HIVE-IR+ highlights the importance of doing QEP selection along with WAN-aware task placement and transfer scheduling. We measure the improvements of CLARINET and HIVE-IR+ in terms of percentage reduction in average query run time compared to HIVE+.

7.1 Testbed Deployment Results

Deployment Setup and Workload: We spin up 5 server instances each with 40 vCPUs (2.4 GHz Intel Xeon Processors) and 160GB RAM in all 10 EC2 regions. We deploy HDFS+YARN across all the instances; a single server in one of the regions functions as the HDFS namenode and the YARN resource manager. The connectivity between different sites is through the public Internet; naturally available bandwidth (see fig. 2) acts as the constrained resource. To avoid disk read/write bottlenecks, we store all the intermediate data in memory; this also aligns with recent trends toward in-memory analytics [48, 27]. We use TPC-DS queries on datasets at different scales (10, 50, 100, 500) for our evaluation. Our workload is generated by randomly choosing the queries and the scale of data. The input tables are randomly spread across the different geographical regions, similar to prior studies [35, 43].

Comparison with single DC execution model: Figure 9 compares running times of individual TPC-DS queries using HIVESINGLEDC and HIVE+. For only 2 of the 24 different queries that we evaluated, HIVESINGLEDC has a smaller running time than HIVE+; further, HIVESINGLEDC can be up to four times slower ($0.25\times$) than HIVE+.

Upon closer investigation, we find that for the queries where HIVESINGLEDC is faster, the distribution of the largest input table was skewed; 70% of the input data was in one DC. Thus, for such cases, HIVESINGLEDC requires only 30% of the mapper outputs and none of the reducer outputs to be transferred across the WAN.

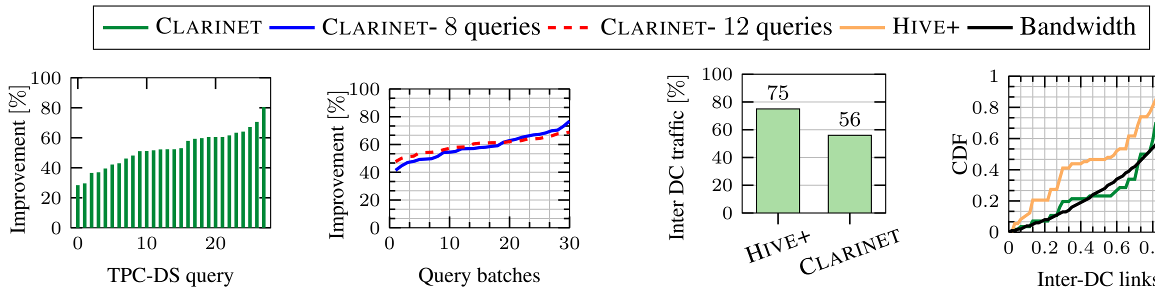
Overall, the distributed execution model effectively utilizes the total WAN bandwidth when the input data is spread across multiple DCs. However, when the input data is skewed, placing all the reducers in one DC is advantageous. Thus, in all further experiments, we also consider a task placement strategy where all the tasks are placed in the DC with the largest input data in addition to the placement approaches discussed in §4.1. CLARINET's design and the iterative heuristic described in §5.1 easily accommodate multiple task placement strategies for each QEP.

Clarinet performance: Figure 10(a), shows the run time reduction of CLARINET compared to HIVE+ for TPC-DS queries when run individually. We can see that network-aware QEP selection, task placement, and scheduling results in at least a 20% reduction (or 1.25x speedup) in query run time; the gains can be as high as 80% (5x) for some of the queries. For 75% of the queries, CLARINET chooses an alternate QEP than the one chosen by default in Hive (not shown). This highlights the importance of network-aware QEP selection even for single queries.

Figure 10(b) shows the gains when multiple TPC-DS queries of different scales are run simultaneously; we report results over 30 randomly chosen batches of TPC-DS queries with 8 and 12 queries in a batch. More than 40% (1.66x) gains are observed in all the batches. On average, we see a ≈60% reduction or 2.5x speedup, higher than the single query case (45% on average).

By placing the reducers randomly across different geographical regions, HIVE+ transfers 75% (Figure 10(c)) of the total intermediate data between EC2 DCs. Since, the inter-region bandwidth is limited, this leads to longer running times. CLARINET on the other hand, transfers only half of the intermediate data between DCs.

Figure 10(d) shows the distribution of bandwidth and intermediate data across the inter-site links for a



(a) Percentage reduction in running times of CLARINET w.r.t. HIVE+ for 29 individual TPC-DS queries. The queries are sorted based on the observed gains.

(b) Percentage reduction in average completion times of CLARINET w.r.t. HIVE+ when batches of 8 / 12 randomly chosen TPC-DS queries of different scales are executed simultaneously. The batches are sorted based on the observed gains.

(c) Comparison of HIVE+ and CLARINET w.r.t. intermediate data sent over the WAN as a percentage of the total intermediate data. The values are measured over a single run with 12 simultaneous queries.

(d) Comparison of bandwidth and intermediate data distribution across a subset of pairwise logical links between the DCs for a single batch of 12 queries. We ignore all links that are unused by both CLARINET and HIVE+.

Figure 10: Results from a real CLARINET deployment across Amazon EC2 datacenters.

single run with 12 simultaneously running queries. The difference between intermediate data and bandwidth distribution is greater for HIVE+ when compared to CLARINET. For example, HIVE+ transfers 45% of its intermediate data over logical links that account for only 20% of the bandwidth. In comparison, CLARINET transfers only 20% of its load on 20% of the bandwidth. By considering multiple candidate QEPs for each query and by controlling task placement and scheduling, CLARINET is able to match intermediate data to available bandwidth across different links. Since HIVE+ does not have alternate choices of QEP and task placement/schedule, it tends to put more load on some links and no load on others.

Multi-query optimization: To quantify the need for multi-query optimization in the geo-distributed setting, we measure how the QEPs chosen for queries when run in a joint manner differ from the QEPs chosen when run individually. For 60% of the queries when run with 8 or 12 queries concurrently, the QEP of choice in CLARINET differs from the one chosen when the queries are run individually. As an illustrative example of CLARINET's cross-query behavior, consider TPC-DS query 7; it involves a five-way join of a fact table with 4 other dimension tables one of which is fairly large. Thus, when run by itself, CLARINET never joins the fact table with a large dimension table (even though they are located in DCs within a continent) to avoid costly WAN transfer. However, in 5 out of 6 batches when Query 7 runs simultaneously with other queries that load links behind the preferred dimension table, CLARINET forces Query 7 to join large tables upfront.

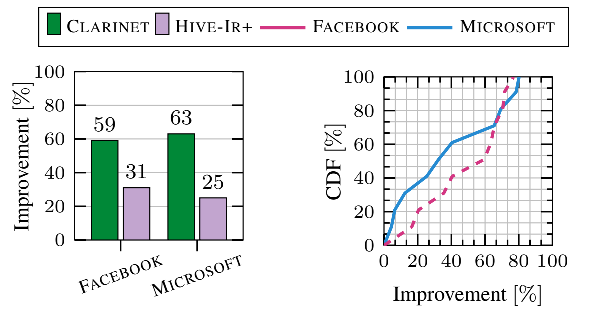
Resource Fragmentation: For a single run with 12 simultaneously running queries, we compute the duration for which inter-DC links remain idle. A resource is *idle* if a task is available to run, but is not scheduled for execution. For CLARINET, *the links are fallow only for 3% of the time*, which is minimal. Our larger scale simulation results confirm reduction in resource fragmentation imposed by our approach.

Optimization overhead: We also measured the time CLARINET spends in optimizing the query plan. After parallelizing the evaluation of each candidate query for every iteration, we see that CLARINET spends less than 1s (on an average) per iteration. For optimizing 12 queries in 30 different batches, CLARINET takes a maximum of 15s; the median optimization time is 8s for a batch of 12 queries. Relative to query execution times (tens of minutes), the optimization overhead of CLARINET is acceptable in practice.

7.2 Simulation Results

Trace driven Simulator: For large-scale experiments, with 50 sites and thousands of queries, we evaluate CLARINET through a trace-driven simulation *based on* production traces obtained from analytics clusters of two large OSPs, FACEBOOK and MICROSOFT. These traces contain information on query arrival times, input data/intermediate data size for each query, data locations, QEP structure etc., for 350K and 600K jobs respectively. Please refer to [37, 18] for more details about the workloads.

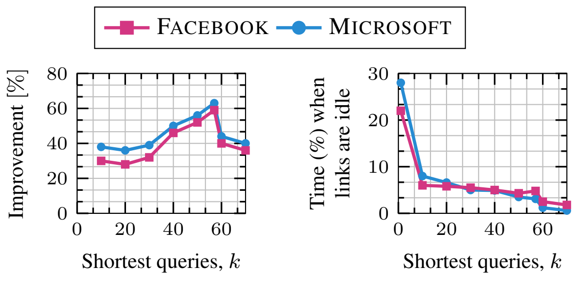
Unfortunately, we do not have logs from the query optimizer that generated the QEP, and hence do not have information regarding alternate QEPs. To overcome this, we use QEPs generated from TPC-DS queries superimposed with information on input table size and intermediate data size from the traces. Thus, every job in the trace is replaced by a randomly chosen TPC-DS query. The TPC-DS input tables acquire the distribution and location characteristics of input data for the corresponding job in the trace. Thus, our workload



(a) Percentage reduction in average running times of CLARINET and HIVE-IR+ w.r.t HIVE+.

(b) CDF of the per-query gains of CLARINET using FACEBOOK and MICROSOFT traces.

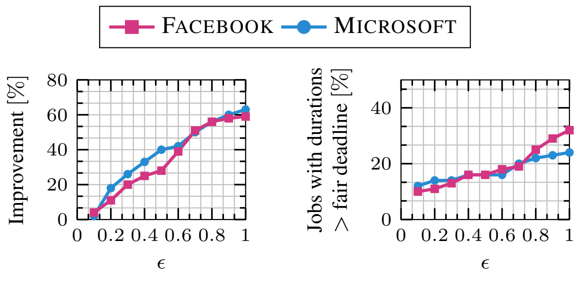
Figure 11: Overall gains of CLARINET and HIVE-IR+ w.r.t HIVE+ as measured in the simulator using FACEBOOK and MICROSOFT production traces



(a) Variation of performance with k, for shortest-k heuristic

(b) Reduction in resource fragmentation with increasing k

Figure 12: Performance of our overall heuristic as a function of k.



(a) Percentage reduction in average query run times relative to HIVE+

(b) Percentage of jobs that do not meet their fair deadline

Figure 13: Variation of performance and fairness metric with respect to ϵ

has similar load and data distributions as the production traces but using query plan options from the TPC-DS benchmark.

Queries arrive in batches of a few hundred; results for other batch sizes are similar. We impose a logical full mesh topology with the bandwidth between each pair of sites chosen randomly from [100Mb/s – 5Gbps].

Figure 11(a) shows the reduction in average running time for CLARINET and HIVE-IR+ when compared to HIVE+ for both the production traces. Compared to the HIVE+ base line, CLARINET improves the average query completion time by 60%, or a 2.5× speedup.

CLARINET offers 28 and 38 percentage points improvement over HIVE-IR+ for FACEBOOK and MICROSOFT traces, respectively. This translates, respectively, to 1.75× and 2× speedup relative to HIVE-IR+. These additional gains come from choosing better QEPs.

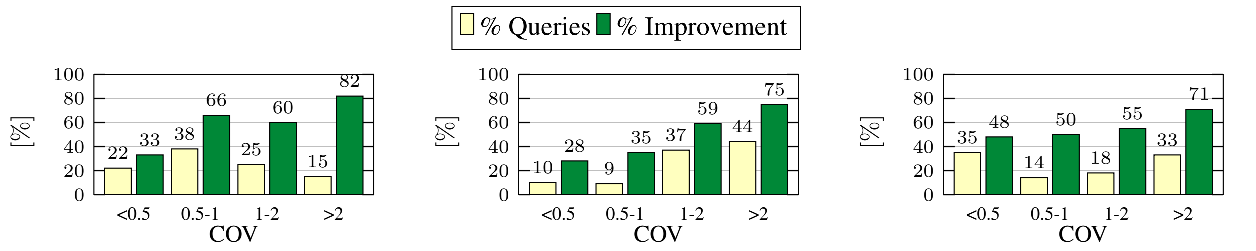
For a network topology with higher bandwidths and low variation (drawn from [10Gbps – 50Gbps]), CLARINET has 47% and 52% reduction in run time for FACEBOOK and MICROSOFT traces, respectively, relative to HIVE+. Higher bandwidth implies overall smaller running times even for a WAN-agnostic system like HIVE+. Even under such a scenario CLARINET offers a 2× improvement.

Figure 11(b) plots the distribution of CLARINET's gains w.r.t HIVE+. Note that CLARINET does not increase the running time for any query. However, the distribution has a heavy tail; some queries have moderate improvement but others have substantial improvement. The variation is especially prominent in MICROSOFT traces, where approximately 38% of the queries have less than 20% (1.25x) improvement and 20% of the queries have greater than 70% improvement (or 3x speedup). In §7.4, we present an in-depth analysis of performance improvement for different classes of queries.

7.3 CLARINET's heuristics and design decisions

Next, we explore the effectiveness of key CLARINET design decisions in simulation.

Effectiveness in Combating Resource Fragmentation: Recall from §5.2 that our approach to combat resource fragmentation is to allow network transfers from top-k shortest queries to be scheduled if resources are fallow. Figure 12(a) plots the variation of overall runtime reduction for different values of k. For $k = 1$, CLARINET does vanilla SJF scheduling. As we increase k, the gain increases, peaks at $k = 57$ for both the FACEBOOK and MICROSOFT traces, and then decreases. The vanilla shortest job is considerably worse than choosing the best value of k. Figure 12(b), shows the fraction of time (in percentage) inter-site links remain fallow as k varies. We see severe underutilization of resources at $k = 1$, explaining the poor performance of SJF. At peak ($k = 57$), we see that the links are not utilized only 5% of the time. Any further increase in k results in decreased link fallow time; however, higher values of k lead to cases where the initial stages (mappers) of k QEPs get scheduled first, as they are available immediately. As a result, resources are blocked for later stages for all k QEPs, resulting in an increase in average run times.



(a) Performance across queries binned by total amount of intermediate data

(b) Performance across queries binned by total input size of tables

(c) Performance across queries binned by bandwidth skew

Figure 14: Isolating gains observed across queries

C: Clarinet CO: Clarinet-O

	Facebook		Microsoft	
	C	CO	C	CO
25%ile	27	13	9	8
Mean	59	30	63	34
75%ile	68	36	67	40
90%ile	72	47	78	48

Table 2: Clarinet vs. a variation allowing overlap of network transfers. Hive+ is used as the baseline.

Fairness across queries: Recall from §5.3 that Clarinet uses a knob ϵ to ensure fairness: $\epsilon \rightarrow 0$ tends to bias Clarinet's core heuristic (§5.2) to schedule from jobs that are nearing deadlines computed based on their fair share (hence leading to greater fairness), whereas $\epsilon \rightarrow 1$ favors performance at the expense of jobs being delayed beyond their fair-share deadlines. Performance improvements from Clarinet relative to Hive+ as a function of ϵ are shown in Figure 13(a). We see that even when biasing toward fairness ($\epsilon \rightarrow 0$), Clarinet offers substantial improvements (20%) relative to Hive+. As we trade-off some amount of fairness (higher ϵ), Clarinet's benefits improve almost linearly. Figure 13(b) shows the percentage of jobs that did not meet the fair deadline as a function of ϵ. For low values of $\epsilon(= 0.1)$, we see that almost 90% of jobs meet their deadline and are not starved by jobs arriving in the future.

Non-overlap: We compare Clarinet with a system, Clarinet-O that disregards Clarinet's schedule and allows tasks to be scheduled as and when they are available. Competing flows on a WAN link will now share the bandwidth equally in space rather than sharing them across time. The QEPs chosen and the placement of tasks are identical in both the cases. Table 2 reports the run time reduction with respect to Hive+. We note that even with an overlapped schedule, the gains of Clarinet-O over Hive+ are significant; average run time reduces by 34% ($1.5\times$). This is due to good QEP selection and task placement. Further, Clarinet is 29 percentage points better than Clarinet-O by virtue of combining QEP selection and task placement with non-overlapped transfer scheduling. Overlap results in lower allocation of bandwidth for all contending flows, thereby increasing all queries' completion times.

7.4 Profiling gains of queries (simulation)

To isolate characteristics of queries that contribute to higher performance, we categorize them based on the amount of skew in (i) the intermediate data generated from different stages, (ii) the spread of input data across sites, and (iii) the average outgoing bandwidth of sites where input tables of a query are located. For each characteristic, we split the queries into "bins" based on the normalized standard deviation (COV). Figure 14 presents the performance gains for queries in each bin.

Queries with high skew (> 2) in the amount of intermediate data perform $3\times$ better than queries for which the intermediate data is equally distributed. The absolute improvement over Hive+ is as high as 82%. A similar trend is observed for queries categorized by the skew in input data. For queries with low skew in intermediate data/input data, all join orders (all possible QEPs) will exercise all links in the topology. Thus, choosing one over another will not offer substantial improvement in performance.

We also observe performance gains improving (48 to 71%) with growing bandwidth skew, but the effect is less pronounced. This is consistent with the high gains observed in a homogeneous WAN substrate (§7.2). Clarinet performance is not intrinsically tied to the presence of high WAN skew.

8 Related Work

We discuss related work on query optimization in §2.2. Clarinet adds to the rich literature on query optimization in both (distributed) database systems [34, 14, 25, 12, 32, 41, 38, 46, 49] and big data analytics stacks [2, 10]. In particular, it shows how to bring WAN awareness into query optimization in a principled fashion.

Other recent work have explored low-layer optimizations to improve GDA query performance. Iridium [35] develops WAN-aware input data and task placement for two-stage MapReduce jobs. Geode [43] develops input data movement and join algorithm selection strategies to minimize WAN bandwidth usage. Finally, Jetstream [36] proposes using adaptive filtering and local aggregation of data to improve latency. SWAG [21] coordinates compute task scheduling across DCs.

Many of these apply to simple 1- or 2-stage queries [35, 21, 43], whereas CLARINET considers general DAGs. Some also require detailed modifications to existing analytics frameworks [36], whereas CLARINET's design is such that it can be integrated with ease. More importantly, CLARINET operates at a higher layer than all prior systems, by optimizing query plan generation. Thus, CLARINET has a more fundamental impact on query performance. Also, CLARINET is complementary to these prior systems (e.g., [21, 36]).

9 Discussion

Experimental results highlight CLARINET's performance achieved through WAN-aware QEP selection, combined with operator placement and scheduling aspects of the execution framework. While this motivates the need to explore non-traditional query optimization approaches for the geo-distributed settings, there are a few other aspects to consider.

First, the efficacy of CLARINET depends on the availability of known, non-fluctuating bandwidth between DCs. Most software-defined WAN managers provide this abstraction under normal operating conditions. Further, our experiments on Amazon EC2 showed that minor fluctuations in available bandwidth do not adversely affect CLARINET's performance. However, under catastrophic network failures, the bandwidth availability between DCs can change drastically. CLARINET does not have any mechanism to react under such scenarios. Prior works [11, 29, 9] have presented approaches to dynamically change query execution plans under system changes and cardinality estimation errors. Developing similar techniques to adapt CLARINET's execution plan under bandwidth changes is part of our future work.

Second, CLARINET does not leverage performance gains obtained from using techniques that minimize the overall data transferred over the WAN. These include: (i) using bloom-filters to implement joins as semi-joins, and (ii) caching (intermediate) results data from prior queries [43]. While reducing WAN traffic improves query completion time in the geo-distributed setting, the total data sent over the WAN (e.g., determined by the number of common keys in a bloom-filter semi-join implementation) can be large depending upon the dataset. Under such cases, network-aware QEP selection and scheduling of transfers can further reduce aggregate run times even if WAN traffic reduction methods are used.

10 Conclusion

In this paper, we consider the problem of running analytics queries over data gathered and stored at multiple sites inter-connected by heterogeneous WAN links. We argue that, in order to optimize query completion times, it is crucial for the query plan to be made WAN-aware, for query planning to be done jointly with selecting the placements and schedule for the query's tasks, and for multiple queries to be jointly optimized. We design CLARINET, a novel WAN-aware QO that incorporates a variety of novel heuristics for these issues. We implement CLARINET such that it can be easily integrated into existing data analytics frameworks with minimal modifications. Our experiments using an EC2 deployment and large scale simulations show that CLARINET reduces query completion times by $2\times$ compared to using state-of-the-art WAN-aware placement and scheduling. We also show how our scheme can ensure fair treatment of queries.

Acknowledgments

We thank the anonymous reviewers and our shepherd Amol Deshpande for their insightful comments. Raajay and Aditya are supported by the Wisconsin Institute on Software-defined Datacenters of Madison and grants from Google and National Science Foundation (CNS-1302041, CNS-1330308, CNS-1345249).

References

[1] Amazon datacenter locations. https://aws.amazon.com/about-aws/global-infrastructure/.

[2] Apache Calcite - a dynamic data management framework. http://calcite.incubator.apache.org. Accessed 04-27-2015.

[3] Apache Hive. http://hive.apache.org.

[4] Apache Tez. http://tez.apache.org.

[5] Google datacenter locations. http://www.google.com/about/datacenters/inside/locations/.

[6] Microsoft datacenters. http://www.microsoft.com/en-us/server-cloud/cloud-os/global-datacenters.aspx.

[7] Spark SQL. https://spark.apache.org/sql.

[8] TPC Benchmark DS (TPC-DS). http://www.tpc.org/tpcds.

[9] AGARWAL, S., KANDULA, S., BRUNO, N., WU, M.-C., STOICA, I., AND ZHOU, J. Reoptimizing data parallel computing. In *NSDI* (2012).

[10] ARMBRUST, M., XIN, R. S., LIAN, C., HUAI, Y., LIU, D., BRADLEY, J. K., MENG, X., KAFTAN, T., FRANKLIN, M. J., GHODSI, A., AND ZAHARIA, M. Spark SQL: Relational data processing in Spark. In *SIGMOD* (2015).

[11] AVNUR, R., AND HELLERSTEIN, J. M. Eddies: Continuously adaptive query processing. In *Proceedings of the 2000 ACM SIGMOD International Conference on Management of Data* (New York, NY, USA, 2000), SIGMOD '00, ACM, pp. 261–272.

[12] Bernstein, P. A., and Chiu, D.-M. W. Using semi-joins to solve relational queries. *Journal of the ACM 28*, 1 (1981), 25–40.

[13] Calder, M., Fan, X., Hu, Z., Katz-Bassett, E., Heidemann, J., and Govindan, R. Mapping the expansion of Google's serving infrastructure. In *IMC* (2013).

[14] DeWitt, D. J., Ghandeharizadeh, S., Schneider, D., Bricker, A., Hsiao, H.-I., Rasmussen, R., et al. The Gamma database machine project. *IEEE Transactions on Knowledge and Data Engineering 2*, 1 (1990), 44–62.

[15] Ganjam, A., Siddiqui, F., Zhan, J., Liu, X., Stoica, I., Jiang, J., Sekar, V., and Zhang, H. C3: Internet-scale control plane for video quality optimization. In *12th USENIX Symposium on Networked Systems Design and Implementation (NSDI 15)* (Oakland, CA, May 2015), USENIX Association, pp. 131–144.

[16] Graefe, G. Volcano: An extensible and parallel query evaluation system. *IEEE Trans. on Knowl. and Data Eng. 6*, 1 (Feb. 1994), 120–135.

[17] Graefe, G. The cascades framework for query optimization. *Data Engineering Bulletin 18* (1995).

[18] Grandl, R., Ananthanarayanan, G., Kandula, S., Rao, S., and Akella, A. Multi-resource packing for cluster schedulers. In *SIGCOMM* (2014).

[19] Gupta, A., Sudarshan, S., and Vishwanathan, S. Query scheduling in multi query optimization. In *Database Engineering and Applications, 2001 International Symposium on.* (2001), IEEE, pp. 11–19.

[20] Hong, C.-Y., Kandula, S., Mahajan, R., Zhang, M., Gill, V., Nanduri, M., and Wattenhofer, R. Achieving high utilization with software-driven WAN. In *SIGCOMM* (2013).

[21] Hung, C.-C., Golubchik, L., and Yu, M. Scheduling jobs across geo-distributed datacenters. In *SoCC* (2015).

[22] Isard, M., Prabhakaran, V., Currey, J., Wieder, U., Talwar, K., and Goldberg, A. Quincy: Fair scheduling for distributed computing clusters. In *SOSP* (2009).

[23] Jain, S., Kumar, A., Mandal, S., Ong, J., Poutievski, L., Singh, A., Venkata, S., Wanderer, J., Zhou, J., Zhu, M., Zolla, J., Hölzle, U., Stuart, S., and Vahdat, A. B4: Experience with a globally-deployed software defined WAN. In *SIGCOMM* (2013).

[24] Jiang, J., Das, R., Ananthanarayanan, G., Chou, P., Padmanabhan, V., Sekar, V., Dominique, E., Goliszewski, M., Kukoleca, D., Vafin, R., and Zhang, H. Via: Improving internet telephony call quality using predictive relay selection. In *SIGCOMM* (2015).

[25] Kitsuregawa, M., Tanaka, H., and Moto-Oka, T. Application of hash to data base machine and its architecture. *New Generation Computing 1*, 1 (1983), 63–74.

[26] Kumar, A., Jain, S., Naik, U., Raghuraman, A., Kasinadhuni, N., Zermeno, E. C., Gunn, C. S., Ai, J., Carlin, B., Amarandei-Stavila, M., Robin, M., Siganporia, A., Stuart, S., and Vahdat, A. BwE: Flexible, hierarchical bandwidth allocation for WAN distributed computing. In *SIGCOMM* (2015).

[27] Li, H., Ghodsi, A., Zaharia, M., Shenker, S., and Stoica, I. Tachyon: Reliable, memory speed storage for cluster computing frameworks. In *Proceedings of the ACM Symposium on Cloud Computing* (New York, NY, USA, 2014), SOCC '14, ACM, pp. 6:1–6:15.

[28] Mackert, L. F., and Lohman, G. M. R* optimizer validation and performance evaluation for distributed queries. In *PVLDB* (1986).

[29] Markl, V., Raman, V., Simmen, D., Lohman, G., Pirahesh, H., and Cilimdzic, M. Robust query processing through progressive optimization. In *Proceedings of the 2004 ACM SIGMOD International Conference on Management of Data* (New York, NY, USA, 2004), SIGMOD '04, ACM, pp. 659–670.

[30] Mastrolilli, M., and Svensson, O. (acyclic) job shops are hard to approximate. In *FOCS* (2008).

[31] Monaldo, M., and Ola, S. Improved bounds for flow shop scheduling. In *ICALP* (2009).

[32] Mullin, J. K. Optimal semijoins for distributed database systems. *IEEE Transactions on Software Engineering 16*, 5 (1990), 558–560.

[33] Olston, C., Reed, B., Srivastava, U., Kumar, R., and Tomkins, A. Pig latin: A not-so-foreign language for data processing. In *Proceedings of the 2008 ACM SIGMOD International Conference on Management of Data* (New York, NY, USA, 2008), SIGMOD '08, ACM, pp. 1099–1110.

[34] Polychroniou, O., Sen, R., and Ross, K. A. Track join: distributed joins with minimal network traffic. In *SIGMOD* (2014).

[35] Pu, Q., Ananthanarayanan, G., Bodik, P., Kandula, S., Akella, A., Bahl, V., and Stoica, I. Low latency geo-distributed data analytics. In *SIGCOMM* (2015).

[36] Rabkin, A., Arye, M., Sen, S., Pai, V. S., and Freedman, M. J. Aggregation and degradation in JetStream: Streaming analytics in the wide area. In *NSDI* (2014).

[37] Ren, X., Ananthanarayanan, G., Wierman, A., and Yu, M. Hopper: Decentralized speculation-aware cluster scheduling at scale. In *Proceedings of the 2015 ACM Conference on Special Interest Group on Data Communication* (New York, NY, USA, 2015), SIGCOMM '15, ACM, pp. 379–392.

[38] Rodiger, W., Muhlbauer, T., Unterbrunner, P., Reiser, A., Kemper, A., and Neumann, T. Locality-sensitive operators for parallel main-memory database clusters. In *ICDE* (2014).

[39] Roy, P., Seshadri, S., Sudarshan, S., and Bhobe, S. Efficient and extensible algorithms for multi query optimization. In *Proceedings of the 2000 ACM SIGMOD International Conference on Management of Data* (New York, NY, USA, 2000), SIGMOD '00, ACM, pp. 249–260.

[40] Sellis, T. K. Multiple-query optimization. *ACM Trans. Database Syst. 13*, 1 (Mar. 1988), 23–52.

[41] Urhan, T., and Franklin, M. J. XJoin: A reactively-scheduled pipelined join operator. *IEEE Data Engineering Bulletin* (2000), 27–33.

[42] Viswanathan, R., Ananthanarayanan, G., and Akella, A. Clarinet: Wan-aware optimization for analytics queries. Tech. Rep. TR1841, University of Wisconsin-Madison, 2016.

[43] Vulimiri, A., Curino, C., Godfrey, B., Padhye, J., and Varghese, G. Global analytics in the face of bandwidth and regulatory constraints. In *NSDI* (2015).

[44] Wang, X., Olston, C., Sarma, A. D., and Burns, R. Coscan: Cooperative scan sharing in the cloud. In *Proceedings of the 2Nd ACM Symposium on Cloud Computing* (2011), SOCC '11.

[45] Xiao, X., Hannan, A., Bailey, B., and Ni, L. M. Traffic engineering with mpls in the internet. *Network, IEEE 14*, 2 (2000), 28–33.

[46] Xin, R. S., Rosen, J., Zaharia, M., Franklin, M. J., Shenker, S., and Stoica, I. Shark: SQL and rich analytics at scale. In *SIGMOD* (2013).

[47] XIONG, P., HACIGUMUS, H., AND NAUGHTON, J. F. A software-defined networking based approach for performance management of analytical queries on distributed data stores. In *SIGMOD* (2014).

[48] ZAHARIA, M., CHOWDHURY, M., DAS, T., DAVE, A., MA, J., MCCAULEY, M., FRANKLIN, M., SHENKER, S., AND STOICA, I. Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. In *NSDI* (2012).

[49] ZAMANIAN, E., BINNIG, C., AND SALAMA, A. Locality-aware partitioning in parallel database systems. In *SIGMOD* (2015).

JetStream: Cluster-scale parallelization of information flow queries

Andrew Quinn, David Devecsery, Peter M. Chen and Jason Flinn
University of Michigan

Abstract

Dynamic information flow tracking (DIFT) is an important tool in many domains, such as security, debugging, forensics, provenance, configuration troubleshooting, and privacy tracking. However, the usability of DIFT is currently limited by its high overhead; complex information flow queries can take up to two orders of magnitude longer to execute than the original execution of the program. This precludes interactive uses in which users iteratively refine queries to narrow down bugs, leaks of private data, or performance anomalies.

JetStream applies cluster computing to parallelize and accelerate information flow queries over past executions. It uses deterministic record and replay to time slice executions into distinct contiguous chunks of execution called epochs, and it tracks information flow for each epoch on a separate core in the cluster. It structures the aggregation of information flow data from each epoch as a streaming computation. Epochs are arranged in a sequential chain from the beginning to the end of program execution; relationships to program inputs (sources) are streamed forward along the chain, and relationships to program outputs (sinks) are streamed backward. JetStream is the first system to parallelize DIFT across a cluster. Our results show that JetStream queries scale to at least 128 cores over a wide range of applications. JetStream accelerates DIFT queries to run 12–48 times faster than sequential queries; in most cases, queries run faster than the original execution of the program.

1 Introduction

Dynamic information flow tracking (DIFT) has emerged as an important tool for understanding and troubleshooting program behavior. Originally proposed by the security community [16], DIFT instruments an application binary to track data and/or control flow from global sources (e.g., program inputs) to global sinks (e.g., program outputs). Information flow analysis has proven to be helpful in a diverse set of domains that include forensic analysis [12], information provenance [6], privacy [7], application debugging [20], and troubleshooting of configurations [1, 2].

Unfortunately, dynamic information flow analysis can be painfully slow; depending on the granularity and amount of information tracked, execution slowdowns of up to one or two orders of magnitude are common. While this cost can be reduced by limiting analysis to managed languages such as Java or by restricting the types of queries that can be performed, *general-purpose* information flow analysis over binary code requires batch-style analysis for substantial programs. In other words, the user employing DIFT must run such analyses over the course of hours. DIFT would be much more powerful if analysis could be employed interactively; for instance, a user could refine a particular query by changing sources, sinks, the propagation function, the granularity of instrumentation, or the period of program execution over which the analysis is employed. The user could then narrow down the bug, misconfiguration, or privacy violation in a manner similar to traditional debugging techniques.

Our goal is to make DIFT queries interactive by parallelizing them across many cores in a compute cluster. With hundreds or thousands of cores, DIFT queries that previously took hours or days can complete in seconds or minutes, enabling refinement and iteration over multiple queries. Thus, a shared cluster can become a valuable resource for a large team of system operators or programmers who want to occasionally engage in an interactive debugging or troubleshooting session using DIFT tools. Usage scenarios for DIFT include both live analysis (as the program runs) and after-the-fact analysis (executed on a replay of an execution). We target the latter scenario.

Previous efforts at parallelizing DIFT have met with only limited success. Information flow is inherently difficult to scale (Ruwase et al. call it "embarrassingly sequential" [20]) because it tracks many fine-grained sequential dependencies between memory and register values. The set of dependencies at each step is a function of a large number of prior instructions executed by the program. Consequently, prior efforts have produced parallel versions that scale to only a few cores on a single machine, and no current approach can effectively leverage a commodity cluster to scale DIFT.

Our solution, JetStream, provides cluster-level scalability by computing information flow in two phases, each using a different form of parallelization.

The first phase is the **local DIFT phase**; it divides program execution into time segments (epochs) and assigns a separate core to compute the information flow for each epoch. Each core determines the dependencies within its epoch that may be relevant to answering the overall query. It tracks sources and sinks that are identified explicitly in the query (e.g., network input and output); we call these *global sources/sinks*. It also tracks locations that may serve as links between global sources and sinks; we call these *local sources/sinks*. Local sources are all memory addresses and registers at the beginning of an epoch, and local sinks are all memory addresses and registers at the end of an epoch. The local DIFT phase parallelizes cleanly into separate partitions, with all dependencies between partitions resolved in the next phase.

Two challenges arise for the local DIFT phase. First, computing all dependencies that may be relevant to the query is too expensive. We address this challenge by deferring and avoiding work as much as possible. JetStream uses merge trees [20] to represent and manipulate dependency sets more efficiently. More importantly, it defers traversing these merge trees until the next (aggregation) phase; that phase avoids traversing the vast majority of tree nodes that do not lie on a dependency path between a global source and a global sink.

The second challenge is that each epoch must follow the same execution as the original execution, so that the aggregation of local DIFTs produces a result equivalent to a sequential DIFT. JetStream uses checkpointing and deterministic record/replay to divide an execution into epochs and perform the local DIFT for each epoch independently, yet consistently. JetStream uses lightweight statistics collected during the original execution of a program to partition the DIFT work equally, and it uses heavyweight statistics collected during the first query of an execution to better partition subsequent queries.

The second phase, called the **aggregation phase**, prunes and combines the information from the local DIFT phase to compute the final result, i.e., the relationship between global sources and global sinks across the entire execution. The cores in this phase are organized in a chain in order of program execution, and the computation is structured as a stream processing algorithm with pipeline-style parallelism. Each core resolves dependencies using information from one epoch's local DIFT phase, and the global query is answered via two streaming passes.

In the first streaming pass, the locations (registers and memory addresses) that are derived from global sources are passed *forward* along the chain (from the beginning to the end of execution). This information is used to prune deferred operations that do not depend on a global source. In the second streaming pass, the locations that propagate dependencies to a sink are passed *backward* along the chain. This lets JetStream prune deferred operations on which no sink depends.

The structuring of the aggregation is the most important factor in enabling JetStream to scale much better than prior approaches at parallelizing DIFT. Our insight is that a small amount of sequential information is necessary to avoid huge amounts of unnecessary work; this information is essentially the locations that depend on global sources (forward pass) and the locations on which global sinks will depend (backward pass). Streaming this data along a sequential chain allows most processing to occur in parallel, with the sequential limitation being essentially the time to pass a single data value from one end of the chain to the other; this is much less than the total query time even for hundreds of processors.

The contributions of this paper are:

- An algorithm for parallelizing DIFT that scales much better than prior approaches, enabling interactive (sub-minute) response times.
- Scalable and efficient support for tracking millions of distinct global sources and sinks at byte granularity, without restrictions on source-code availability, compute platform, or query type.
- A detailed evaluation of the remaining bottlenecks in accelerating DIFT through parallelization.

We have applied JetStream to run DIFT queries over seven desktop and server applications: Evince, Firefox, Ghostscript, Gzip, Mongodb, Nginx, and OpenOffice. Our results show that JetStream scales DIFT to at least 128 cores for these applications. It accelerates DIFT queries to run 12–48 times faster than sequential queries, and, in most cases, runs queries faster than the original execution of the program.

2 Motivation

DIFT is a fundamental analysis that is useful in diverse domains. For example, Arnold [6] uses DIFT for provenance queries that reveal how data values in files and application memory were derived. In forensics [12], DIFT has been used to answer questions such as: "How was my system compromised?" and "What data was leaked?" TaintDroid [7] and similar systems use DIFT to reveal whether an application execution leaks sensitive data. X-Ray [1] uses DIFT to identify misconfigurations that cause performance anomalies, and ConfAid [2] uses DIFT to identify misconfigurations that cause bugs. Poirot's [11] use of DIFT helps determine if a security vulnerability has been exploited.

Many of the above systems run complex DIFT queries on native binaries and can suffer from painfully slow DIFT query times. These systems are often forced to use batch-style computation, even though many would ideally be interactive in nature.

Consider a developer debugging an incorrect output value from a Web server. Using JetStream, she begins by running a DIFT query that shows all program inputs from which the faulty value was derived. This alone is not enough to reveal the bug, so she runs an additional query tracking the inputs that led to a correct output. Comparing the results shows that inputs from a particular network connection led to the faulty output but not the correct one. Using this information, she discovers a bug in the code which parses network inputs. To see if this bug has impacted any file system state, the developer runs another query specifying all values from the faulty parsing code as global sources and all file system outputs as global sinks. She detects that no permanent state has been affected by her bug. Next she considers other forms of external output such as network messages.

Debugging the problem and determining the impact of the bug both require multiple DIFT queries. Further, phrasing the correct queries may be non-trivial and require multiple iterations to get helpful results. If each query takes hours to complete, then this process only makes sense for the most difficult bugs. In contrast, low-latency DIFT enables information flow analysis to be an integral part of the debugging process.

3 Background

We first describe two technologies on which JetStream builds: dynamic information flow tracking and deterministic record and replay.

3.1 DIFT

Dynamic information flow tracking, sometimes referred to as taint tracking, instruments applications to monitor data flow as programs execute. In its most general form, DIFT reveals which *global sources* causally affect which *global sinks* according to a *propagation function*. Global sources are typically external program inputs, such as bytes read from a file or a network socket, and global sinks are typically external outputs.

The propagation function specifies what information flows to track during program execution. For example, a basic data flow propagation function for the instruction $x = y + z$ would state that the sources on which x depends are the union of the sources on which y and z depend. Usually, DIFT tracks data flow (as we do in this work), but some DIFT systems also track implicit flows propagated via control flow.

When an application executes, DIFT assigns a *taint identifier* to each unique global source. For each location, it maintains a set of taint identifiers that shows the global sources on which that location currently depends, and it updates taint sets as instructions execute. At each global sink, DIFT outputs the set of taint identifiers of all locations written to the sink (e.g., the bytes sent to a network socket). Thus, DIFT produces a set of $\langle globalsource, globalsink \rangle$ tuples that describe how particular global sources and global sinks are related.

JetStream tracks global sources, global sinks, and dependencies at byte granularity using binary instrumentation inserted by Pin [14]. A single JetStream query may look for relationships between millions of distinct global sources and sinks. In contrast, many prior DIFT systems require source code or the use of a managed language runtime. Others track only whether any global source data propagates to a global sink and cannot determine *which* sources affect each sink—such systems cannot answer questions such as: "Which inputs affected this program value?" or "What data did I leak?"

A JetStream query contains a program execution to monitor, a filter that specifies the global sources, a filter that specifies sinks, and a propagation function. For instance, a provenance query [6] might wish to determine the lineage of the data in a particular file. The source filter would match all external program inputs and the output filter would match writes to a particular file. This would reveal which bytes in the file were derived from which sources. Alternatively, a privacy query [7] might specify reads from sensitive files as sources and network outputs as sinks. This would reveal what data was leaked over the network and how it was leaked. JetStream provides an interface for supporting custom propagation functions and supplies Arnold's copy, data, and index propagation functions [6] as defaults.

For complex applications, mapping all global sources to all global sinks at byte granularity produces far too much information (e.g., terabytes of data for some benchmarks in Section 5). Thus, filters are needed to extract the right information succinctly. This leads to refinement through iteration. Our goal is to make DIFT fast enough to be interactive, so that a user can issue multiple queries to search for the right information.

3.2 Deterministic replay

Deterministic replay allows the execution of a program to be recorded and reproduced faithfully. When a program first executes, all inputs from nondeterministic actions are logged; these values are supplied during subsequent replays in lieu of performing the nondeterministic operations again. Thus, the program starts in the same state, executes the same instructions on the same data values, and generates the same results.

JetStream derives several benefits from using deterministic replay. First, replay allows JetStream to partition a recorded execution into epochs and execute these epochs in parallel. Deterministic replay guarantees that the result of stitching together all epochs is equivalent to a sequential execution of the program. Second, replay allows an execution recorded on one machine to be replayed on a different machine. There are few external dependencies, since interactions with the operating sys-

	Instructions	Taint IDs	Merge Log	Live Set (forward pass)	Taint Tuples (backward pass)
Epoch 0	1. A = read() 2. B = read() 3. C = A + B	A: IN_0 B: IN_1 C: $M_0[0]$	$M_0[0] : \{IN_0, IN_1\}$	↓ $\{A,B,C\}$	$\{< IN_0, OUT_0 >, < IN_1, OUT_0 >\}$ ↑
Epoch 1	4. D = X + Y 5. E = C 6. B = 0 7. Z = A[D]	D: $M_1[0]$ E: C_1 B: {} Z: $M_1[1]$	$M_1[0] : \{X_1, Y_1\}$ $M_1[1] : \{A_1, M_1[0]\}$	↓ $\{A,C,E,Z\}$	$\{< OUT_0, C >\}$ ↑
Epoch 2	8. F = E 9. write(F)	F: E_2 OUT_0: E_2		↓	$\{< OUT_0, E >\}$ ↑

Table 1: DIFT analysis of an example program.

tem and other external entities are nondeterministic and replayed from the log. Thus, the only requirement for replay is that the replaying computer has the same hardware architecture as the recording computer and that it runs a kernel modified to support replay. Finally, replay allows iterative queries over the same execution.

JetStream uses Arnold [6] to provide deterministic record and replay of multithreaded, multiprocess applications. Arnold's performance overhead is less than 10% for most workloads, and its storage overhead is reasonable even for continuous recording of a workstation.

4 Design and implementation

To parallelize a DIFT query, JetStream divides an execution into epochs and assigns each epoch to a different core. JetStream then evaluates the query in two phases: a *local DIFT* phase and an *aggregation* phase.

In the local DIFT phase, each core concurrently computes the relationships between sources and sinks within its epoch. A core can directly observe global sources and global sinks that occur during its epoch. However, some locations at the start of an epoch may depend on global sources from preceding epochs, and the local DIFT cannot know the actual dependencies because the local DIFTs for those preceding epochs are being executed concurrently. Thus, for all epochs but the first one, the local DIFT phase conservatively tracks all locations at the start of the epoch as *local sources* and assigns a unique *local source identifier* to each location at the epoch start. Similarly, the local DIFT cannot determine which locations at the end of an epoch will ultimately propagate to global sinks in succeeding epochs, so the local DIFT treats all locations at the end of the epoch as *local sinks*. A local DIFT phase thus tracks and reports dependencies between all sources (both local and global) and all sinks (both local and global).

In the aggregation phase, the cores organize as a chain in program execution order and communicate local DIFT results forward and backward along the chain to produce the final set of $\langle globalsource, globalsink \rangle$ tuples. We next describe these two phases in more detail.

Table 1 shows an example query in which JetStream finds all dependencies from global sources to global sinks in a simple program. Program execution is divided into three epochs (shown by the horizontal partitioning). The local DIFT phase is the region to the left of the double vertical bar, and the aggregation phase is the region to the right of the bar. Instructions 1 and 2 read data from global sources, and instruction 9 writes to a global sink.

4.1 Local DIFT

JetStream implements the local DIFT phase as a Pin tool. Executing an application with this tool attached is quite slow (e.g., 14–75x slowdown for the benchmarks in Section 5). There are two reasons: DIFT may add several additional instructions to track taint for each application instruction executed, and Pin dynamically adds the instrumentation to an application as it executes. The first is a fundamental cost of DIFT, while the second is a consequence of using a dynamic instrumentation tool.

The local DIFT phase for a given epoch first replays the application uninstrumented to advance its execution to the start of the epoch, a process we call *fast forwarding*. JetStream may start the replay from the beginning, or it may start from a checkpoint of application state taken during recording or during a previous query. Given the relative speed difference between instrumented and uninstrumented execution, starting from the beginning is reasonable for low numbers of epochs. As the number of epochs increases, fast forward time comes to dominate total query time, and checkpoints are quite beneficial.

Next, JetStream attaches the DIFT tool to the application and Pin starts instrumenting the application to track dependencies. JetStream assigns a unique *source identifier* to each location modified by a global source in the epoch and to each location at the start of the epoch.

JetStream runs all threads of a multithreaded application on a single core to realize an important performance benefit: the instrumentation code does not need to obtain locks to synchronize access to the DIFT data structures because only one thread runs at any given time [21]. JetStream still fully utilizes the processor because each core runs a different epoch in parallel.

For each location, JetStream stores an integer *taint identifier* that represents the set of global and local sources on which that location currently depends. A taint identifier may be: (1) a global source identifier, (2) a local source identifier, or (3) an identifier that maps to a set of global and local sources. In the Taint IDs column of Table 1, IN_0 and IN_1 are global source identifiers, and C_1 and E_2 are local source identifiers that represent the taint of address C at the start of epoch 1 and the taint of address E at the start of epoch 2.

For each x86 instruction, the local DIFT tool reads the taint identifiers of the instruction's inputs and updates the taint identifiers of the instruction's outputs. Taint identifiers for registers are stored in a per-thread array, and taint identifiers for memory addresses are stored in a two-level page table. The tool decomposes the work for each instruction into a sequence of four sub-commands: `set`, `clear`, `copy`, and `merge`. The first three sub-commands are straightforward—`set` assigns a taint identifier to a location, `clear` assigns the NULL identifier to a location, and `copy` sets the destination's taint identifier equal to the source's. Thus, each of these sub-commands are low overhead integer operations. Table 1's Taint IDs column shows how the local DIFT tool updates the taint data structures: a `set` for instruction 1, a `clear` for instruction 6, and a `copy` for instruction 5.

The `merge` sub-command is used for instructions that combine dependencies, e.g., instructions 3, 4, and 7. For these instructions, the set of sources on which the output depends is the union of the sets of sources on which the inputs depend. Our original implementation tracked such sets explicitly, but this worked poorly. For some complex applications, the DIFT did not finish after running for hours, or the size of the sets exceeded the 256 GB memory of our server. Intuitively, the reason is that the set of tuples that relate all local sources to all local sinks can be as large as the size of the address space squared.

We therefore turned to an idea proposed by Ruwase et al. [20] in which sets of taint values are represented by a binary tree. Each `merge` operation generates a new taint identifier to represent the set union. JetStream writes an entry to a *merge log*, which contains the taint identifiers of the input to the `merge`. Thus, the merge log is a DAG sorted in temporal order, and each node (entry) in the merge log represents a binary tree of taint identifiers rooted at that node. Any merge node can be *resolved* to a set of source identifiers by performing a depth-first traversal of the tree rooted at that node.

In the example, instruction 4 creates a merge node $M_1[0]$ (each epoch has a distinct merge log, with the particular log denoted by the subscript). The node states that address D depends on whatever X and Y depend on at the start of epoch 1. Instruction 7 creates a merge node that has $M_1[0]$ as a child, so address X depends on whatever A, X, and Y depend on at the start of the epoch.

Using the merge log yields two benefits. First, it defers expensive set union operations until the aggregation phase; optimizations in that phase avoid the need to perform the vast majority of such unions. Second, the merge log uses much less memory than storing a set for each location. Memory usage is roughly proportional to the number of unique merge operations rather than the total size of all taint sets for every location. The cost of using a merge log is that JetStream must perform a tree traversal when it needs to resolve a root node to a set of source identifiers.

JetStream makes two enhancements to Ruwase et al's algorithm. First, it uses a hash table to cache recently-seen merge pairs and reuse merge nodes when duplicates are found. Second, whereas Ruwase et al. used the tree data structure only for abstract values (i.e., local source identifiers); JetStream also uses the tree structure for sets of global source identifiers, such as distinct bytes from different sources encountered during the local epoch (as for instruction 3 in the example).

At the end of an epoch, JetStream writes four datasets to a shared memory buffer: global source metadata, global sink metadata, the merge log, and the taint identifiers for all local sinks. The global source metadata describes each global source identifier (e.g., the system call that read the byte, the file the byte was read from, the offset within the file, etc.). Similarly, the global sink metadata describes each byte sent to a sink. Since application execution typically modifies only a small percentage of locations during a given epoch, the local sink identifiers for most locations will be the local source identifier of those locations. To save space, the local DIFT only outputs those local sink identifies where this relationship does not hold. These optimizations allow the output of the local DIFT phase to fit in the memory of modern servers (though it is still large, e.g., a few GB per epoch).

4.2 Partitioning

The time to produce an answer to a query depends on the longest local DIFT time for any epoch. Thus, to achieve good speedups, JetStream must partition local DIFT so that each core does roughly the same amount of work. To accomplish this, JetStream estimates the amount of time it will take to run local DIFT for any given interval of execution and defines epoch boundaries so that the estimated local DIFT time for each epoch is the same.

We estimate the local DIFT time for an interval of execution as a linear combination of three factors:

- **Fast Forward Time:** JetStream replays the application without instrumentation to advance execution to the start of the epoch. We estimate that this component of work is proportional to the user-level CPU time used for this portion of execution by the recorded application.

- **Instructions executed:** To track information flow for an interval of execution, JetStream must execute the instructions in that interval, as well as the instrumentation code that propagates dependencies among locations. This component of work is proportional to the number of instructions executed, which we estimate from the user-level CPU time used to execute the interval in the recorded execution.

- **Unique instructions executed:** Pin instruments an instruction when it is executed for the first time. With Pin, instrumentation cost is a significant portion of the overall DIFT time, especially for short intervals in which each instruction may only be executed a few times. As JetStream parallelizes the DIFT work across more cores, each interval becomes shorter, and the relative cost of instrumenting instructions increases. This component of work is proportional to the number of *unique* instructions executed. During recording, we read processor performance counters via the `perf_events` API to estimate the number of unique instructions executed by sampling the instruction pointer (we sample every 32 L1 instruction cache read misses for user-level code). When executing the first query for an execution, we use dynamic instrumentation to measure the actual number of unique instructions executed during an interval; this adds little overhead compared to DIFT instrumentation.

To avoid confounding testing and training in our evaluation, we choose the constants in the model for the first query of an execution by running a linear regression over data from the *other* benchmarks in our set. The coefficient of determination (R^2 value) for these regressions is 0.86–0.87. Due to the high overhead of instrumenting code with Pin, the cost of inserting instrumentation (proportional to unique instructions executed) usually dominates the cost of running the instrumented code (proportional to instructions executed), especially for small epochs.

For subsequent queries of a given execution, we run a linear regression over the performance data gathered during the first query. This produces a much better R^2 value of 0.985. We also add the number of merges that occurred during each interval to our model, and that change slightly increases the R^2 value to 0.989.

JetStream partitions the recorded execution into n epochs of roughly equal local DIFT time as estimated by the above model, where n is the number of available cores to run the query. This process is conceptually simple, but a complication is that the total local DIFT time depends on the particular partitioning chosen because an instruction that is executed in multiple epochs will incur an instrumentation cost in each of those epochs. We solve this problem by using a hill-climbing algorithm in which each iteration updates the estimate of the total local DIFT time for the query, and the new estimate is used to calculate a better partitioning in the next iteration. Usually, this process converges after a small number of iterations.

4.3 Aggregation

The aggregation phase produces the set of $\langle globalsource, globalsink\rangle$ tuples that are related by the propagation function. Within a single epoch, a global source and global sink are related if the global sink either has the global source's identifier or if it has the identifier of a merge node and that node resolves to a set that contains the global source's identifier. If the global source and sink are in adjacent epochs, then they are related if there exists a location L at the boundary of the two epochs such that, in the first epoch, the local sink identifier of L depends on the global source, and in the second epoch, the global sink depends on the local source identifier of L. If one or more epochs separate the epochs of the global source and sink, then there must be multiple such relationships forming a continuous path from source to sink.

In Table 1, such a path exists between the global sources of instruction 1 and 2 and the global sink of instruction 9. In epoch 0, resolving the merge tree for address C reveals that it depends on both global source 0 and global source 1. In epoch 1, the final value of E depends on the value of C at the beginning of the epoch. In epoch 2, instruction 9 writes address F, which depends on location E at the beginning of the epoch. Determining the complete relationship between global sources and global sinks requires aggregating the data from the local DIFT phase of each epoch.

4.3.1 Parallelizing aggregation: A failed attempt

To meet our performance goals, both the local DIFT and aggregation phases must scale well with the number of cores. Our first approach to constructing a parallel aggregation phase was based on a tree-like merge of local DIFT information. First, each individual epoch produces a map of all $\langle source, sink\rangle$ tuples where sources and sinks may be either local or global. For each sink with a taint identifier that represents a merge node, the map is generated by a depth-first traversal of the tree rooted at that node to resolve the set of source identifiers. This step is performed in parallel for all epochs, and caching is used to avoid revisiting tree nodes. If both the source and sink in a tuple are global, then the tuple is immediately output and removed from the map. Such tuples represent dependencies that can be computed solely on the DIFT information in a local epoch.

Next, we merge maps for pairs of adjacent epochs. For all locations, L, if there exists a tuple $\langle source, L\rangle$ in the first epoch and a tuple $\langle L, sink\rangle$ in the second epoch, then this step adds the tuple $\langle source, sink\rangle$ to its map. Since two epochs are involved, this step is parallelized across two cores. As before, if the sources and sinks in a tuple are both global, the tuple is immediately output and removed from the map. Merges are performed in a binary tree, merging sets of 4, 8, 16, etc. epochs, using the same approach as above. The number of cores participating in each merge grows proportionally, and the number of merge steps is logarithmic in the number of epochs.

Unfortunately, this algorithm performed very poorly. Traversing the merge log for each end value and generating sets of start values was extremely time-consuming, even with caching and reuse of intermediate results. Even worse, for most of our applications, some of the merged maps failed to fit in 256 GB of memory. Our analysis showed that the reason for this behavior was that we were doing far more work than we needed to: the vast majority of merge nodes visited and values in the merged maps were not actually on a path between a global source and a global sink. However, because no epoch knew the full set of global sources and sinks when creating or merging maps, each had to calculate all dependencies that could possibly be used.

We concluded from this failed attempt that a fully-parallel aggregation phase is infeasible because it vastly increases the total work done. To fix this, aggregation must use data about global sources and sinks to generate less intermediate data and to traverse fewer merge nodes.

4.3.2 Backward pass

Our next approach to aggregation was to structure the computation as a stream processing algorithm that scales via pipeline-style parallelism. We arrange the epochs in an ordered chain. In parallel, each epoch processes any global sinks encountered during the epoch. The JetStream aggregator checks the taint identifier for each byte sent to a global sink. If the taint identifier is a global source (i.e., if the global source and sink are in the same epoch), the aggregator immediately outputs a $\langle globalsource, globalsink\rangle$ tuple. If the taint identifier is a local source identifier L, the aggregator sends a $\langle L, globalsink\rangle$ tuple to the *previous* epoch in the chain. Epochs on the same machine communicate via a shared memory buffer; epochs on different machines communicate via a TCP socket. If the taint identifier is a merge log node, the aggregator resolves the set with a depth-first traversal of the tree rooted at that node. For each unique source identifier in the set, it either outputs a $\langle globalsource, globalsink\rangle$ tuple or sends a $\langle L, globalsink\rangle$ tuple to the previous epoch.

When the aggregation phase for an epoch receives a $\langle L, globalsink\rangle$ tuple from the succeeding epoch, it checks the epoch's local sink taint identifier for L. This is either a local source identifier, a global source identifier, or a merge node identifier that resolves to a set of source identifiers. For each global source, the aggregator outputs a $\langle globalsource, globalsink\rangle$ tuple, and for each local source L', it sends a $\langle L', globalsink\rangle$ tuple to the preceding epoch.

The last epoch sends a sentinel value to its preceding epoch after it has finished processing its sinks; when an epoch reads the sentinel, its work is done as no more tuples will be forthcoming. It then sends the sentinel to its predecessor.

The last column of Table 1 shows the backward pass. Epoch 2 determines that OUT_0 depends on location E and passes that tuple to epoch 1. Epoch 1 determines that E depends on C, so passes the tuple $\langle OUT_0, C\rangle$ to epoch 0. Epoch 0 resolves the merge tree rooted at $M_0[0]$ and outputs tuples relating OUT_0 with both IN_0 and IN_1.

The major advantage of this streaming algorithm is that no epoch will process a merge node or send a tuple to a proceeding epoch unless the node/tuple represents a location that propagates to some global sink according to the propagation function. In the example, no merge nodes in epoch 1 are visited. This vastly reduces the amount of aggregation work. The potential disadvantage of this algorithm is that we have added a sequential step; each tuple must flow from global sink to global source, passing through all intermediate epochs. Our results show that this has only a minor effect on overall query time since each core can still process tuples in parallel. In other words, the latency of passing a tuple through all epochs in the chain is very small compared to the query time, just as the sequential time to execute a machine instruction in a pipelined CPU is trivial compared to the time spent operating with a full pipeline.

Our results show that this algorithm, which we will refer to as the *backward pass* produces reasonable aggregation costs for some simple applications/queries, but it still takes too long for complex applications/queries. The reason is that we are still visiting too many merge nodes and creating too many tuples that are not ultimately on the path between a source and a sink. Thus, we found it necessary to also add a streaming *forward pass* that propagates information about which locations are related to global sources along the chain of epochs.

4.3.3 Forward pass

The forward pass runs prior to the backward pass. For each epoch, the forward pass first calculates a *reverse index* that has the same vertexes as the merge log DAG but that has edges in the opposite direction. The reverse index is also a DAG; depth-first traversal from a given local or global source yields the set of sinks that depend on that source. Each epoch builds its reverse index by first visiting all merge log nodes in temporal order, then visiting all local sinks. This step is fully parallelized since the reverse index can be computed purely with local information for each epoch.

Next, for each byte read from a global source, the aggregator does a depth-first traversal of the reverse index to determine the set of local sinks that depend on any source. It passes these sink locations to the succeeding epoch in the chain (forward in time). Here, the aggregator is only determining that a given local sink is tainted by any global source; it is not identifying a particular global source that has tainted the local sink. Therefore, the aggregator passes a local sink to the succeeding epoch at most once, and it visits each node in the reverse index at most once. It sets a `visited` bit for each local sink and merge node to avoid duplicate work.

As the aggregator receives locations from the prior epoch, it does a depth-first traversal of the reverse index to determine which (if any) additional local sinks depend on that location. It sends the locations associated with those local sinks to the succeeding epoch. The aggregator also retains the complete set of locations obtained from the prior epoch; this *live set* is the set of all local sources that depend on any global source.

Similar to the backward pass, the first epoch sends a sentinel token as soon as it finishes processing global sources. Once an epoch receives the sentinel, its live set is complete; the epoch then sends the sentinel to its successor.

In Table 1, the Live Set column shows the forward pass. At the end of the first epoch, locations A, B, and C depend on at least one global source. The second epoch adds Z to this set because it depends on A and adds E to this set because it depends on C. Additionally, the second epoch removes B from this set because its taint value was cleared.

Once an epoch knows its live set, it prunes its merge log. The aggregator processes merge log nodes sequentially. Any local source not in the live set for that epoch cannot depend on a global source. So, if a child of a merge node is a local source identifier, and the local source is not in the live set, the child is replaced by a NULL identifier. If a merge node has two NULL children, no members of its source set depend on a global source. Any identifier in the merge log that refers to such a node is also replaced with a NULL value. Essentially, this is a garbage collection in which any node known to be unrelated to a global source is removed. This garbage collection can substantially prune the merge log. Each epoch can run the prune in parallel once it knows its live set. Thus, the only sequential component of the forward pass is the propagation of live set values. In Table 1, epoch 1 prunes merge node $M_1[0]$ because it does not depend on any global source. $M_1[1]$ is updated to $\langle A_1, NULL\rangle$.

By inserting a forward pass, JetStream guarantees that all merge nodes processed and all tuples generated during the backward pass are on a path between a global source and global sink. This vastly reduces the number of nodes processed and tuples generated, making the backward pass more efficient. Note that although the forward pass itself must visit all nodes tainted by a global source (even those that do not lead to a global sink), the forward pass does much less work than the backward pass because it tracks only whether or not a location depends on a global source. It does not identify the specific source(s) on which the location depends.

4.3.4 Pre-pruning

JetStream uses one final optimization to improve aggregation performance. During an epoch, many values in memory or registers are overwritten before the epoch ends. If a merge log node does not propagate to either a local or global sink, then it can be removed from the log based solely on information available from that epoch. We call this step *pre-pruning*. JetStream does pre-pruning via a mark-and-sweep garbage collection over the merge log. It iterates through all sinks; if a sink has the taint identifier of a merge log node, JetStream marks the merge node as referenced. Then, JetStream iterates backward through the merge log. For each child in a merge log entry that refers to a prior merge log node, JetStream marks the prior merge log node as referenced. It discards all unmarked nodes and compacts the merge log. This reduces the number of merge log nodes that need to be processed later during both the forward and backward passes.

Benchmark	Replay Log Size (MB)	Replay Time (seconds)	Sequential DIFT Time (seconds)	Global Sources	Global Sinks	Dependencies
Gzip	0.03	2.98	109.23	64352941	48791393	36586765
Ghostscript	0.12	1.03	76.90	2514067	176009	14682254
Evince	2.90	13.47	234.30	10302852	104061604	346305
Nginx	30.65	4.75	196.51	10412627	35000000	5000000
Mongodb	37.02	22.79	309.99	8863855	116592809	76042962
OpenOffice	15.25	7.55	418.03	9946659	32110959	14599069
Firefox	24.80	67.42	1838.70	920029	1636119	131476

Table 2: Benchmarks used in the evaluation.

4.3.5 Summary

For each epoch, JetStream performs the following operations: (1) It runs the program without instrumentation from the start or from the nearest checkpoint to the beginning of the epoch. (2) It attaches a Pin tool and performs a local DIFT until the end of the epoch. (3) It pre-prunes the resulting local DIFT output to eliminate merge log nodes that cannot lead to a global sink. (4) It performs a forward aggregation pass to further prune the merge log by excluding any node that does not depend on a global source. (5) It performs a backward aggregation pass to generate $\langle globalsource, globalsink \rangle$ tuples; only merge nodes and locations on the path between a source and a sink are visited during this pass. Almost all of these steps can be performed in parallel for each epoch. The exceptions are the propagation of source dependencies in the forward path and $\langle location, sink \rangle$ tuples in the backward pass. These sequential steps are structured as stream processing along the epoch chain to maximize the work done in parallel.

5 Evaluation

Our evaluation answers the following questions:

- How well does JetStream scale DIFT?
- What are the remaining scalability bottlenecks?
- What is the impact of query optimizations?

5.1 Experimental Setup

JetStream uses the Arnold record and replay system [6] and the Pin dynamic instrumentation framework [14]. We evaluated JetStream using a CloudLab [19] cluster of 32 r320 machines (8-core Xeon E5-2450 2.1 GHz processors, 16 GB RAM, 10 Gb NIC). We envision running JetStream on an even larger cluster, but we could only reliably get a 32 machine cluster from CloudLab. Since these machines have a relatively small amount of RAM (16 GB) and DIFT queries are memory-intensive, we use only 4 cores per machine, leaving the experimental setup with 128 effective cores. For all experiments, we report the mean of 5 trials and show 95% confidence intervals.

5.2 Benchmarks

We evaluate JetStream with seven benchmarks chosen to represent common desktop and server workloads:

- **Gzip** – Zip a large file.
- **Ghostscript** – Convert a research poster from PostScript to PDF.
- **Evince** – Open and view a research paper.
- **Mongodb** – Yahoo cloud server benchmark [5].
- **Nginx** – Serve static content.
- **OpenOffice** – Edit a conference presentation.
- **Firefox** – A long Facebook browsing session.

For Gzip, Ghostscript, Evince, Mongodb, and Nginx, the query asks for dependencies between all command line, network, and file system inputs and all such outputs. Running the all-to-all query for OpenOffice and Firefox generated over 1 TB of data before we stopped the query. Thus, the OpenOffice query only considers file system data from the user's home directory to be sources, and the Firefox query considers cookie data to be sources and network output to specific sites (about 10% of total output) to be sinks. We use Arnold's data flow propagation function for all queries.

Table 2 shows a summary of the benchmarks. These are complex queries: most consider millions of distinct source and sinks, and most generate millions of dependencies. We show the time to replay each benchmark without instrumentation and the sequential DIFT time on a replay as baselines. We do not show the time for the original benchmark to run because that time depends on user think-time (for interactive applications), network delays, idle time (for server applications) and external output. When the benchmark is not CPU bound, DIFT overheads can be underestimated. Replay time, against which we compare, can already be one or two orders of magnitude faster than the original execution time [6]. We also report the compressed replay log size for each benchmark.

5.3 Scalability

We first evaluate the scalability of JetStream queries. Figure 1 shows the speedup of executing the first query for each benchmark on a log-log scale as we vary the number of cores from 1 to 128. Results are normalized to evaluating a query using a sequential algorithm on a single core; the black diagonal line shows ideal speedup, and a horizontal line would show no speedup. Overall, JetStream accelerates DIFT queries by 8–28x with

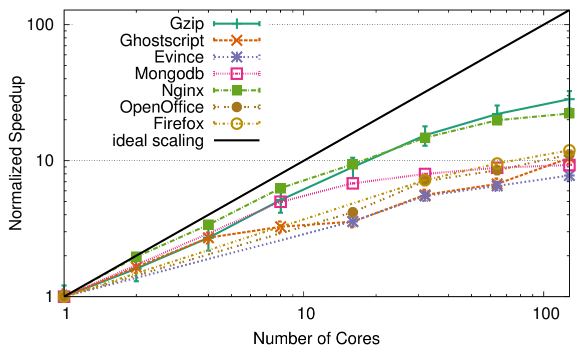


Figure 1: First Query Scalability - JetStream's scalability from 1 to 128 cores

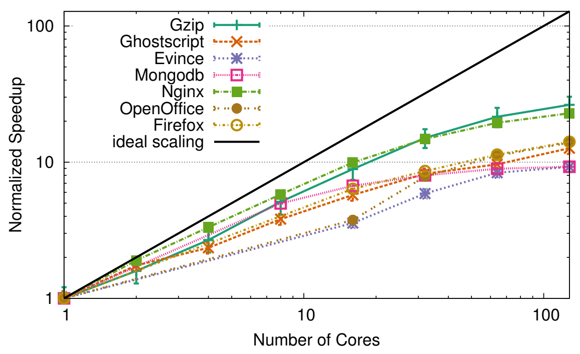


Figure 2: Scalability After Repartitioning - JetStream's scalability after incorporating its improved partitioning model

a mean of 13x using 128 cores. All benchmarks continue to scale through 128 cores, but some (e.g., Evince) scale less well at high numbers of cores.

We find that the biggest bottlenecks to scalability of the first query are (1) the epoch partitioning is often imbalanced, resulting in delays due to tail latency, and (2) the fast forward time becomes a bottleneck as query time approaches the replay time (since we need to replay the application from the beginning to start an epoch).

We address the first bottleneck by gathering data about unique instructions executed during the first query and improving the partitioning. Figure 2 shows the impact of repartitioning for the second query. With this optimization, JetStream scales the DIFT queries by 9–26x, with a mean of 14x. Repartitioning improves performance for all benchmarks except Gzip; in the case of Gzip, the model generated with less-detailed statistics is actually a better predictor of performance than the model generated with more-detailed statistics.

We address the second bottleneck by taking intermediate checkpoints during the first query. Figure 3 shows the scalability of the second query when using both repartitioning and checkpointing. JetStream scales the DIFT queries by 12–48x, with a mean of 21x. All benchmarks continue to scale up to 128 cores, though the pace of scaling diminishes with larger number of epochs. At 128 cores, the Gzip and Mongodb queries execute

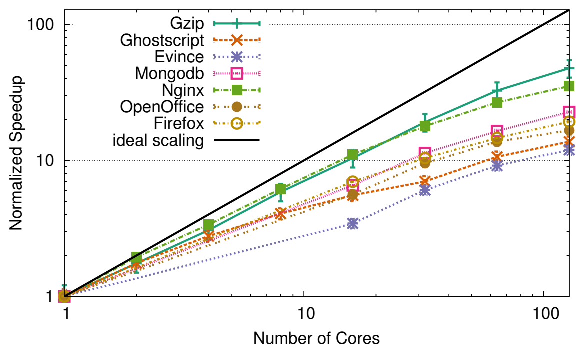


Figure 3: Second Query Scalability - JetStream's scalability after improving partitioning and checkpointing

faster than their sequential replay times, and all benchmarks except Ghostscript execute faster than the original execution time of the application.

5.4 Analysis of first-query bottlenecks

Next, we examine results for individual benchmarks in more detail and identify scalability bottlenecks. Figures 4 and 5 show stacked bar graphs for each benchmark at 128 cores; results for the first query are in the left column, and results for the second query are in the right column.

Each stacked bar in a graph shows the time spent in different query stages for a single epoch; the epochs are ordered left to right by the order of the time slices in the application execution. The bottom region, labeled fast forward, shows the time for application execution to reach the start of the epoch. Instrumentation is the time required for Pin to instrument instructions, and analysis is the time to execute that instrumentation. The split between these two values is estimated by assuming that the instrumentation cost is equal to the unique instructions executed term from the model in Section 4.2 (which has an R^2 value of 0.989) as we cannot directly distinguish these two values.

Pre-prune, forward pass, prune, and backward pass show the time spent in each aggregation stage. The sequential constraints of the forward pass and backward pass are shown by the gently sloping lines at the top of each region: one epoch's forward or backward pass cannot complete until the prior epoch in that pass has completed. The total time to complete the query is given by the height of the first stacked bar; the first epoch is the last to complete aggregation because of the sequential nature of the backward pass.

Outlier epochs caused by the result of poor partitioning can be detected by variance in the tops of the analysis regions (the combination of the fast forward, instrumentation, and analysis phases). All of our benchmarks except Gzip and Mongodb noticeably benefit from improved partitioning. For example, comparing the first and second queries of OpenOffice (Figures 5a and 5b)

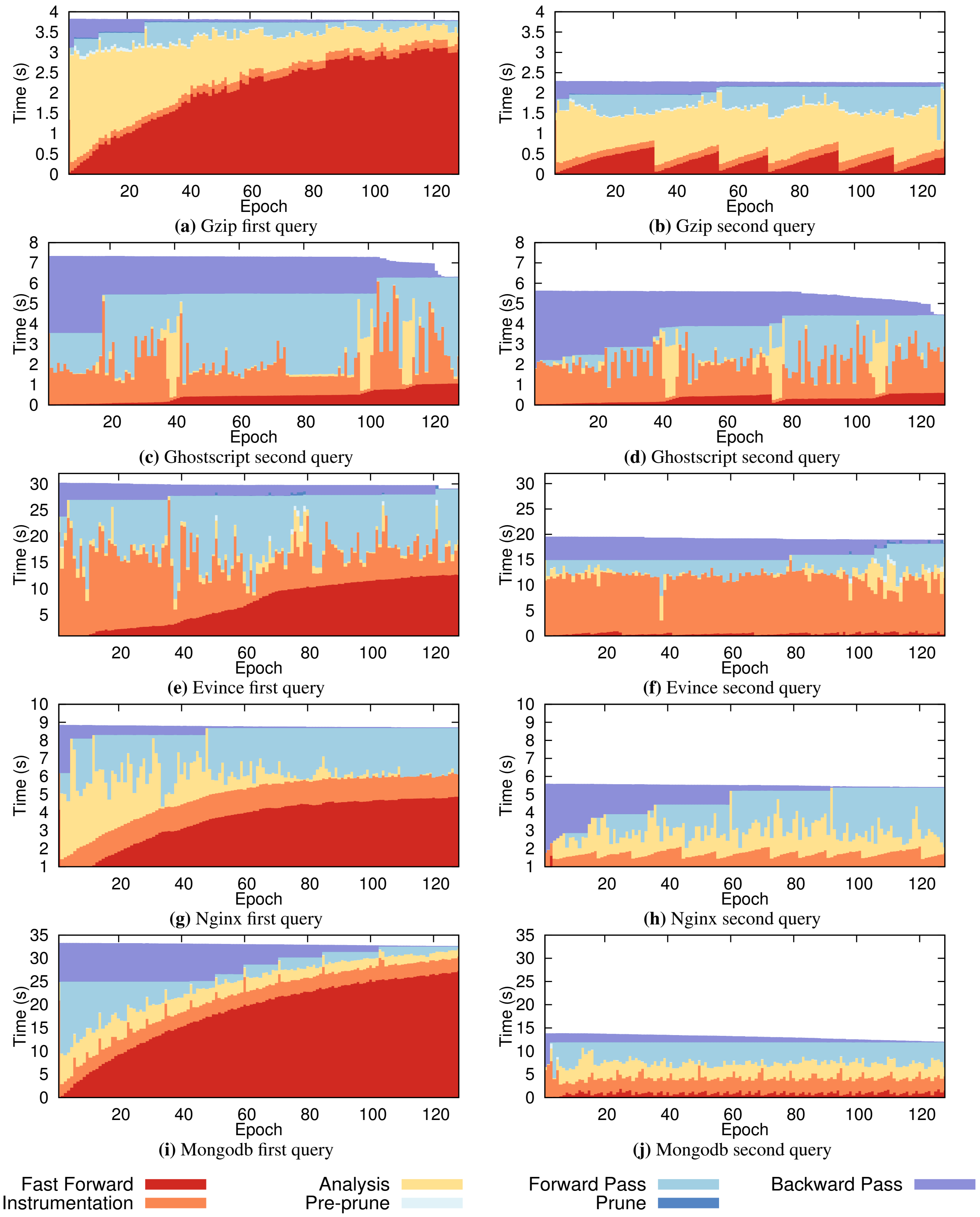
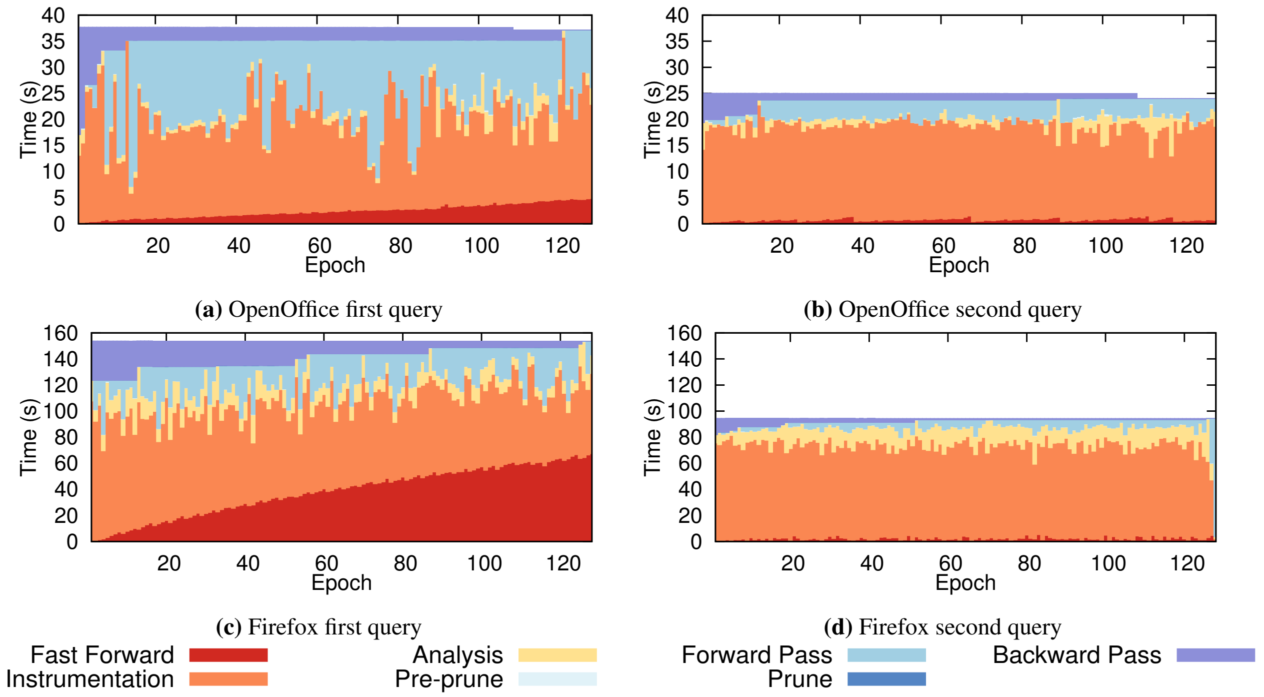


Figure 4: Breakdown of query processing time for 128 cores. Each stacked bar shows one core processing an epoch. From bottom to top, the shaded regions within each bar show the time spent fast forwarding, doing dynamic instrumentation, running DIFT, pre-pruning, performing the forward pass, pruning the merge log and performing the backward pass.



(a) OpenOffice first query **(b)** OpenOffice second query **(c)** Firefox first query **(d)** Firefox second query

Figure 5: Breakdown of query processing time for 128 cores. Each stacked bar shows one core processing an epoch. From bottom to top, the shaded regions within each bar show the time spent fast forwarding, doing dynamic instrumentation, running DIFT analysis, pre-pruning, performing the forward pass, pruning the merge log and performing the backward pass.

shows the benefit of improving the partitioning between the first and second queries. Reducing outliers leads to substantially faster second query times for these benchmarks.

The effects of checkpointing in JetStream can be seen in all of our benchmarks. For example, comparing the first and second queries of Gzip (Figures 4a and 4b) shows the dramatic effect that checkpointing can have on query latency. The primary reason that this benchmark does not scale well for the first query is that the query time approaches the replay time of the benchmark—this is shown by fast forward being a large component of the last epoch time for the first query. In contrast, the fast forward times in the second query are much smaller.

5.5 Analysis of second-query bottlenecks

We next look at second query performance and bottlenecks. Interestingly, the specific bottlenecks vary from benchmark to benchmark.

For Evince (Figure 4f), OpenOffice (Figure 5b), and Firefox (Figure 5d), Pin instrumentation time dominates total query time. Pin instrumentation time also impacts Ghostscript (Figure 4d) to a lesser degree. To explore this issue in more detail, Figure 6 shows the speedup for just instrumentation and analysis. All benchmarks scale up to 128 cores, but not ideally.

There are two main factors that limit instrumentation and analysis scalability: (1) JetStream must taint all local

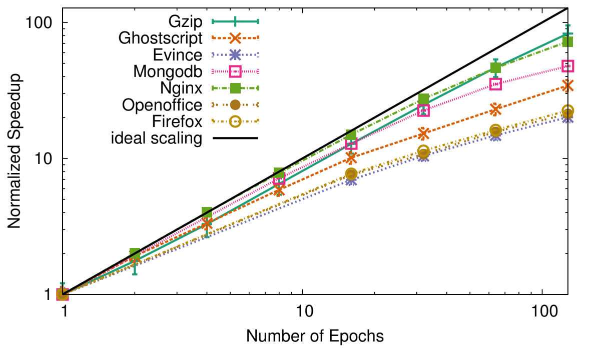


Figure 6: DIFT Scaling - Scalability of local DIFT (excluding fast-forward time) for different numbers of cores normalized to local DIFT (excluding fast-forward time) for one core.

sources at each epoch boundary, and (2) Pin instruments an instruction in every epoch in which that instruction occurs, so dividing the program into smaller epochs increases the total instructions instrumented. We isolated the cost of (1) by running a sequential query on one core that retaints each address at epoch boundaries. This does the exact same work as the parallel version, and it produces the same results; however, Pin instruments each instruction only once across all epochs. As Figure 7 shows, the overhead added by tainting local sources is relatively small (3–25% of the sequential DIFT query). When this overhead is parallelized over 128 cores, it should have little effect on query time. Additionally, our model from Section 4.2 shows that unique instructions correlate very

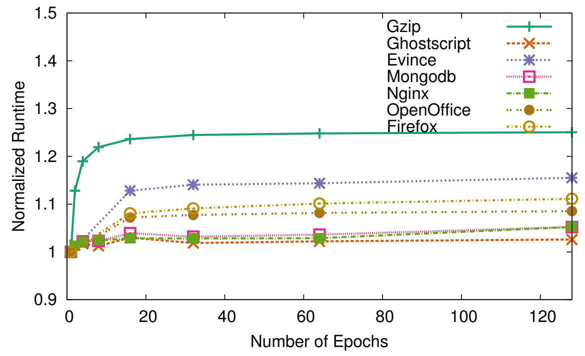


Figure 7: Retainting Overhead - Overhead of tainting local sources at the beginning of each epoch.

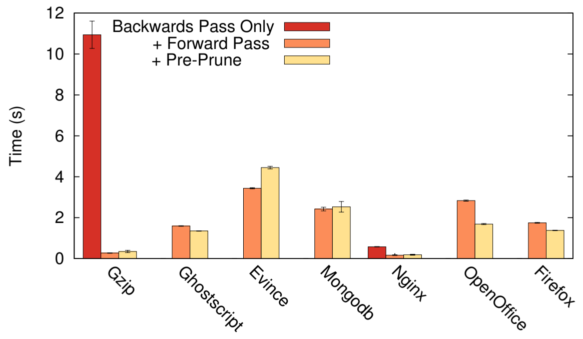


Figure 8: Optimization Effectiveness - Effect of aggregation optimizations at 128 cores. Experiments that did not complete are left blank.

highly with instrumentation and analysis time.

Switching from a dynamic to static instrumentation tool, using techniques that reduce the amount of dynamic instrumentation [8], or employing a low-overhead binary instrumentation platform (e.g., Protean code [13]) could reduce instrumentation time. Alternatively, we could take checkpoints that include already-instrumented code, as is done by Speck [17].

Poor partitioning is a significant component of overall query time for Ghostscript (Figure 4d), Nginx (Figure 4h), and Mongodb (Figure 4j). There are two separate reasons for poor partitioning in these benchmarks.

JetStream gathers statistics about query execution after each system call is executed. Ghostscript contains long regions of computation without a system call. At 128 epochs, JetStream must split some of these regions into multiple epochs. It divides these regions crudely based on the number of entries in the replay log; this crude metric often mispredicts the actual query execution time for the split epochs. Gathering statistics at finer-grained intervals would reduce outliers.

Outlier epochs in Nginx and Mongodb occur due to variance in the amount of taint processing done in each epoch. Outliers are correlated with large numbers of tainted sources and/or sinks in the epoch. Currently, our partitioning tool cannot determine which sources and sinks will be tainted in advance, but JetStream could potentially gather more statistics about sources and sinks during the first query, which could help the partitioning tool make such a determination for subsequent queries.

Aggregation plays a minor role in query time for most benchmarks. The speed of the forward pass is seen in the slope of the top of this region across epochs. Similarly, the speed of the backward pass is given by the slope of the top of that region. The total area for these two regions is less relevant since the sequential constraints mean that epochs will sometimes be idle waiting for data to arrive from predecessor epochs. We see negligible forward pass time across all benchmarks. Backward pass time is most noticeable for Ghostscript (Figure 4d), Mongodb (Figure 4j) and OpenOffice (Figure 5b), as shown by the noticeable slope of the top region in each graph. Better caching heuristics may improve the backward pass for these benchmarks.

5.6 Optimizations

We next evaluate the costs and benefits of optimizations employed by JetStream. We first measure the benefit of two aggregation optimizations: the forward pass and pre-pruning. To isolate aggregation cost from outliers in the local DIFT stage, we let all epochs finish local DIFT before beginning aggregation. This is the worst case for aggregation costs since individual epochs cannot pre-prune or construct the reverse index while waiting for prior epochs to finish local DIFT.

Figure 8 shows the isolated cost of aggregation for 128 cores. If aggregation performs only the backward pass (omitting the forward pass and pre-pruning), then only Gzip and Nginx complete; aggregation runs out of memory on all other benchmarks. For Gzip and Nginx, adding the forward pass improves isolated aggregation time by 98% and 71%, respectively.

The pre-prune optimization appears less effective. It decreases isolated aggregation time for Firefox, OpenOffice, and Ghostscript, but increases it slightly for Mongodb, Evince, and Gzip. We conclude that JetStream's policy of always pre-pruning is likely suboptimal; an adaptive policy that only pre-prunes when spare CPU cycles are available would be better.

We also measured the extra costs to optimize partitioning. We measured the time to profile L1 instruction cache misses for CPU-intensive benchmarks (Gzip and Ghostscript); the average overhead was 3.1%. The average overhead imposed by taking checkpoints during the first query was only 0.7% since each epoch takes at most one checkpoint. Finally, the average overhead of tracing unique instructions during the first query was 1.5%.

6 Related work

JetStream is the first system to parallelize DIFT across a cluster, and it is the first system to efficiently track millions of global sources, global sinks, and dependencies. Several prior systems have parallelized DIFT across the cores of a single machine. To achieve cluster-level scalability, JetStream's main contribution is parallelizing the aggregation of local DIFT data while minimizing the communication between cores.

Like JetStream, Speck [17] partitions an execution into epochs and performs local DIFT for each epoch. Speck tracks only a single label (tainted or untainted). Speck's local DIFT produces a log of sub-commands, which it then optimizes to achieve an up to 6x reduction in log size. Aggregation is done sequentially over the optimized log. This limits the speedup achieved by Speck to only 2x on a 8-core machine.

Ruwase et al. [20] partition an execution into epochs and perform local DIFT on each core using custom hardware [4]. JetStream's merge log optimization is derived from this work; thus, the local DIFT phases of the two systems are similar. However, Ruwase et al. perform aggregation sequentially, and this limits scalability. Like Speck, their system tracks only a single label. TaintPipe [15] partitions DIFT into epochs and tracks taint as symbolic formulas inside each epoch. TaintPipe also performs aggregation sequentially. It is unclear how symbolic tracking can scale efficiently to millions of labels and dependencies.

JetStream focuses on after-the-fact analysis, while prior DIFT parallelization has focused on live analysis during execution. Live analysis runs only a single predefined query, but it is suitable for security use cases in which sensitive actions such as sending network output need to be blocked based on the DIFT results (Speck and Ruwase et al. delay output to support this functionality, while TaintPipe does not). In contrast, after-the-fact analysis is suitable for tasks like forensics [12], debugging [20], configuration troubleshooting [1, 2], analysis of privacy leaks [7], and provenance [6]. No prior system has parallelized after-the-fact DIFT.

Many systems have explored how to make DIFT itself faster. One promising idea is decoupled execution, in which the DIFT work is split into an instrumentation thread and an analysis thread. ShadowReplica [8] combines decoupled execution with static analysis to reduce the amount of instrumentation that Pin must perform. TaintPipe combines decoupled execution with another form of static analysis: taint abstractions for commonly used function. libdft [10] provides several low-level optimizations for accelerating Pin-based DIFT. Profiling and/or static analysis can also reduce the cost of dynamic instrumentation [3, 9, 18].

These ideas are orthogonal to the speedups that JetStream provides through parallelization. In fact, our evaluation shows that Pin dynamic instrumentation is often the scalability bottleneck after JetStream parallelization, so incorporating these optimizations into JetStream is a very promising direction for future work.

7 Conclusion

JetStream enables interactive DIFT over past executions by parallelizing queries across a cluster. It uses deterministic record and replay to divide an execution into epochs and execute a local DIFT for each epoch on a separate core. It aggregates results from local DIFTs by arranging epochs in a sequential chain according to the order of program execution and using a pipeline-like stream processing algorithm to pass information about global sources and sinks along the chain. For future work, we plan to explore novel debugging and forensics applications enabled by JetStream.

Acknowledgments

We thank the anonymous reviewers and our shepherd, Shan Lu, for their thoughtful comments. We also thank Mike Chow and Xianzheng Dou for their help understanding the Arnold code base. We thank the CloudLab team which helped us deploy our system on their experimental platform. This work has been supported by the National Science Foundation under grants CNS-1513718 and CNS-1421441. Any opinions, findings, conclusions, or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the National Science Foundation.

References

[1] Mona Attariyan, Michael Chow, and Jason Flinn. X-ray: Automating root-cause diagnosis of performance anomalies in production software. In *Proceedings of the 10th Symposium on Operating Systems Design and Implementation*, Hollywood, CA, October 2012.

[2] Mona Attariyan and Jason Flinn. Automating configuration troubleshooting with dynamic information flow analysis. In *Proceedings of the 9th Symposium on Operating Systems Design and Implementation*, Vancouver, BC, October 2010.

[3] Walter Chang, Brandon Streiff, and Calvin Lin. Efficient and extensible security enforcement using dynamic data flow analysis. In *Proceedings of the 15th ACM Conference on Computer and Communications Security (CCS)*, Alexandria, VA, October 2008.

[4] Shimin Chen, Michael Kozuch, Theodoros Strigkos, Babak Falsafi, Phillip B. Gibbons, Todd C. Mowry, Vijaya Ramachandran, Olatunji Ruwase, Mchiael Ryan, and Evangelos Vlachos. Flexible hardware acceleration for instruction-grain program monitoring. In *Proceedings of the 35th International Symposium on Computer Architecture (ISCA)*, Beijing, China, June 2008.

[5] Brian F Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. Benchmarking cloud serving systems with ycsb. In *Proceedings of the 1st ACM symposium on Cloud computing*, pages 143–154, 2010.

[6] David Devecsery, Michael Chow, Xianzheng Dou, Jason Flinn, and Peter M. Chen. Eidetic systems. In *Proceedings of the 11th Symposium on Operating Systems Design and Implementation*, Broomfield, CO, October 2014.

[7] William Enck, Peter Gilbert, Byung gon Chun, Landon P. Cox, Jaeyeon Jung, Patrick McDaniel, and Anmol N. Sheth. TaintDroid: An information-flow tracking system for realtime privacy monitoring on smartphones. In *Proceedings of the 9th Symposium on Operating Systems Design and Implementation*, Vancouver, BC, October 2010.

[8] Kangkook Jee, Vasileios P. Kermerlis, Angelos D. Keromytis, and Georgios Portokalidis. ShadowReplica: Efficient parallelization of dynamic data flow tracking. In *Proceedings of the 20th ACM Conference on Computer and Communications Security (CCS)*, Berlin, Germany, November 2013.

[9] Kangkook Jee, Georgios Portokalidis, Vasileios P. Kermerlis, Soumyadeep Ghosh, David I. August, and Angelos D. Keromyrtis. A general approach for efficiently accelerating software-based dynamic data flow tracking on commodity hardware. In *Proceedings of the 19th Network and Distributed System Security Symposium*, San Diego, CA, February 2012.

[10] Vasileios P. Kemerlis, Georgios Portokalidis, Kangkook Jee, and Angelos D. Keromytis. Libdft: Practical dynamic data flow tracking for commodity systems. In *Proceedings of the 8th ACM SIGPLAN/SIGOPS Conference on Virtual Execution Environments*, VEE '12, 2012.

[11] Taesoo Kim, Ramesh Chandra, and Nickolai Zeldovich. Efficient patch-based auditing for Web application vulnerabilities. In *Proceedings of the 10th Symposium on Operating Systems Design and Implementation*, Hollywood, CA, October 2012.

[12] Samuel T. King and Peter M. Chen. Backtracking intrusions. In *Proceedings of the 19th ACM Symposium on Operating Systems Principles*, pages 223–236, Bolton Landing, NY, October 2003.

[13] Michael A Laurenzano, Yunqi Zhang, Lingjia Tang, and Jason Mars. Protean code: Achieving near-free online code transformations for warehouse scale computers. In *Proceedings of the 47th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 558–570, 2014.

[14] Chi-Keung Luk, Robert Cohn, Robert Muth, Harish Patil, Artur Klauser, Geoff Lowney, Steven Wallace, Vijay Janapa Reddi, and Kim Hazelwood. Pin: Building customized program analysis tools with dynamic instrumentation. In *Proceedings of the ACM SIGPLAN 2005 Conference on Programming Language Design and Implementation*, pages 190–200, Chicago, IL, June 2005.

[15] Jiang Ming, Dinghao Wu, Gaoyao Xiao, Jun Wang, and Peng Liu. TaintPipe: Pipelined symbolic taint analysis. In *Proceedings of the 24th Usenix Security Symposium*, Washington, D.C., August 2015.

[16] James Newsome and Dawn Song. Dynamic taint analysis: Automatic detection, analysis, and signature generation of exploit attacks on commodity software. In *Proceedings of the 12th Annual Network and Distributed System Security Symposium*, February 2005.

[17] Edmund B. Nightingale, Daniel Peek, Peter M. Chen, and Jason Flinn. Parallelizing security checks on commodity hardware. In *Proceedings of the 13th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 308–318, Seattle, WA, March 2008.

[18] Feng Qin, Cheng Wang, Zhenmin Li, Ho seop Kim, Yuanyuan Zhou, and Youfeng Wu. Lift: A low-overhead practical information flow tracking system for detecting general security attacks. In *The 39th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO '06)*, Orlando, FL, 2006.

[19] Robert Ricci, Eric Eide, and The CloudLab Team. Introducing CloudLab: Scientific infrastructure for advancing cloud architectures and applications. *USENIX ;login:*, 39(6), December 2014.

[20] Olatunji Ruwase, Phillip B. Gibbons, Todd C. Mowry, Vijaya Ramachandran, Shimin Chen,

Michael Kozuch, and Michael Ryan. Parallelizing dynamic information flow tracking. In *Symposium on Parallelism in Algorithms and Architectures (SPAA)*, June 2008.

[21] Benjamin Wester, David Devescery, Peter M. Chen Jason Flinn, and Satish Narayanasamy. Parallelizing data race detection. In *Proceedings of the 18th International Conference on Architectural Support for Programming Languages and Operating Systems*, Houston, TX, March 2013.

Just Say NO to Paxos Overhead: Replacing Consensus with Network Ordering

Jialin Li Ellis Michael Naveen Kr. Sharma Adriana Szekeres Dan R. K. Ports
University of Washington
{lijl, emichael, naveenks, aaasz, drkp}@cs.washington.edu

Abstract

Distributed applications use replication, implemented by protocols like Paxos, to ensure data availability and transparently mask server failures. This paper presents a new approach to achieving replication in the data center without the performance cost of traditional methods. Our work carefully divides replication responsibility between the network and protocol layers. The network orders requests but *does not* ensure reliable delivery – using a new primitive we call *ordered unreliable multicast* (OUM). Implementing this primitive can be achieved with near-zero-cost in the data center. Our new replication protocol, *Network-Ordered Paxos* (NOPaxos), exploits network ordering to provide strongly consistent replication without coordination. The resulting system not only outperforms both latency- and throughput-optimized protocols on their respective metrics, but also yields throughput within 2% and latency within 16 μs of an unreplicated system – providing replication without the performance cost.

1 Introduction

Server failures are a fact of life for data center applications. To guarantee that critical services always remain available, today's applications rely on fault-tolerance techniques like state machine replication. These systems use application-level consensus protocols such as Paxos to ensure the consistency of replicas' states. Unfortunately, these protocols require expensive coordination on every request, imposing a substantial latency penalty and limiting system scalability. This paper demonstrates that replication in the data center need not impose such a cost by introducing a new replication protocol with performance within 2% of an unreplicated system.

It is well known that the communication model fundamentally affects the difficulty of consensus. Completely asynchronous and unordered networks require the full complexity of Paxos; if a network could provide a totally ordered atomic broadcast primitive, ensuring replica consistency would become a trivial matter. Yet this idea has yielded few gains in practice since traditional ordered-multicast systems are themselves equivalent to consensus; they simply move the same coordination expense to a different layer.

We show that a new division of responsibility between the network and the application can eliminate nearly all replication overhead. Our key insight is that the communication layer should provide a new *ordered unreliable multicast* (OUM) primitive – where all receivers are guaranteed to process multicast messages in the same order, but messages may be lost. This model is weak enough to be implemented efficiently, yet strong enough to dramatically reduce the costs of a replication protocol.

The ordered unreliable multicast model enables our new replication protocol, *Network-Ordered Paxos*. In normal cases, NOPaxos avoids coordination entirely by relying on the network to deliver messages in the same order. It requires application-level coordination only to handle dropped packets, a fundamentally simpler problem than ordering requests. The resulting protocol is simple, achieves near-optimal throughput and latency, and remains robust to network-level failures.

We describe several ways to build the OUM communications layer, all of which offer net performance benefits when combined with NOPaxos. In particular, we achieve an essentially zero-overhead implementation by *relying on the network fabric itself to sequence requests*, using software-defined networking technologies and the advanced packet processing capabilities of next-generation data center network hardware [10, 49, 59]. We achieve similar throughput benefits (albeit with a smaller latency improvement) using an endpoint-based implementation that requires no specialized hardware or network design.

By relying on the OUM primitive, NOPaxos avoids all coordination except in rare cases, eliminating nearly all the performance overhead of traditional replication protocols. It provides *throughput within 2% and latency within 16 μs of an unreplicated system*, demonstrating that there need not be a tradeoff between enforcing strong consistency and providing maximum performance.

This paper makes four specific contributions:

1. We define the *ordered unreliable multicast* model for data center networks and argue that it strikes an effective balance between providing semantics strong enough to be useful to application-level protocols yet weak enough to be implemented efficiently.

2. We demonstrate how to implement this network model in the data center by presenting three implementations:

(1) an implementation in P4 [9] for programmable switches, (2) a middlebox-style prototype using a Cavium Octeon network processor, and (3) a software-based implementation that requires no specialized hardware but imposes slightly higher latency.

3. We introduce NOPaxos, an algorithm which provides state machine replication on an ordered, unreliable network. Because NOPaxos relies on the OUM primitive, it avoids the need to coordinate on every incoming request to ensure a total ordering of requests. Instead, it uses application-level coordination only when requests are lost in the network or after certain failures of server or network components.

4. We evaluate NOPaxos on a testbed using our OpenFlow/Cavium prototype and demonstrate that it outperforms classic leader-based Paxos by 54% in latency and 4.7× in throughput. It simultaneously provides 42% better latency and 24% better throughput than latency- and throughput-optimized protocols respectively, circumventing a classic tradeoff.

2 Separating Ordering from Reliable Delivery in State Machine Replication

We consider the problem of *state machine replication* [56]. Replication, used throughout data center applications, keeps key services consistent and available despite the inevitability of failures. For example, Google's Chubby [11] and Apache ZooKeeper [24] use replication to build a highly available lock service that is widely used to coordinate access to shared resources and configuration information. It is also used in many storage services to prevent system outages or data loss [8, 16, 54].

Correctness for state machine replication requires a system to behave as a linearizable [23] entity. Assuming that the application code at the replicas is deterministic, establishing a single totally ordered set of operations ensures that all replicas remain in a consistent state. We divide this into two separate properties:

1. **Ordering**: If some replica processes request *a* before *b*, no replica processes *b* before *a*.

2. **Reliable Delivery**: Every request submitted by a client is either processed by all replicas or none.

Our research examines the question: *Can the responsibility for either of these properties be moved from the application layer into the network*?

State of the art. Traditional state machine replication uses consensus protocol – e.g., Paxos [33, 34] or Viewstamped Replication [42, 48] – to achieve agreement on operation order. Most deployments of Paxos-based replicated systems use the Multi-Paxos optimization [34] (equivalent to Viewstamped Replication), where one replica is the designated leader and assigns an order to requests. Its normal operation proceeds in four phases: clients submit requests to the leader; the leader assigns a sequence number and notifies the other replicas; a majority of other replicas acknowledge; and the leader executes the request and notifies the client.

These protocols are designed for an *asynchronous network*, where there are no guarantees that packets will be received in a timely manner, in any particular order, or even delivered at all. As a result, the application-level protocol assumes responsibility for both ordering and reliability.

The case for ordering without reliable delivery. If the network itself provided stronger guarantees, the full complexity of Paxos-style replication would be unnecessary. At one extreme, an atomic broadcast primitive (i.e., a *virtually synchronous* model) [6, 27] ensures *both* reliable delivery and consistent ordering, which makes replication trivial. Unfortunately, implementing atomic broadcast is a problem equivalent to consensus [14] and incurs the same costs, merely in a different layer.

This paper envisions a middle ground: an *ordered but unreliable* network. We show that a new division of responsibility – providing ordering in the network layer but leaving reliability to the replication protocol – leads to a more efficient whole. What makes this possible is that an ordered unreliable multicast primitive can be implemented efficiently and easily in the network, yet fundamentally simplifies the task of the replication layer.

We note that achieving reliable delivery despite the range of possible failures is a formidable task, and the end-to-end principle suggests that it is best left to the application [15, 55]. However, ordering without a guarantee of reliability permits a straightforward, efficient implementation: assigning sequence numbers to messages and then discarding those that arrive out of sequence number order. We show in §3 that this approach can be implemented at almost no cost in data center network hardware.

At the same time, providing an ordering guarantee simplifies the replication layer dramatically. Rather than agree on *which* request should be executed next, it needs to ensure only all-or-nothing delivery of each message. We show that this enables a simpler replication protocol that can execute operations without inter-replica coordination in the common case when messages are not lost, yet can recover quickly from lost messages.

Prior work has considered an asynchronous network that provides ordering and reliability in the common case but does not guarantee either. Fast Paxos [36] and related systems [29, 45, 50] provide agreement in one fewer message delay when requests usually arrive at replicas in the same order, but they require more replicas and/or larger quorum sizes. Speculative Paxos [53] takes this further by having replicas speculatively execute operations without

	Paxos [33, 34, 48]	**Fast Paxos** [36]	**Paxos+batching**	**Speculative Paxos** [53]	**NOPaxos**
Network ordering	No	Best-effort	No	Best-effort	Yes
Latency	4	3	4+	2	2
Messages at bottleneck	$2n$	$2n$	$2 + \frac{2n}{b}$	2	2
Quorum size	$> n/2$	$> 2n/3$	$> n/2$	$> 3n/4$	$> n/2$
Reordering/Dropped packet penalty	low	medium	low	high	low

Table 1: Comparison of NOPaxos to prior systems.

coordination, eliminating another message delay and a throughput bottleneck at the cost of significantly reduced performance (including application-level rollback) when the network violates its best-effort ordering property. Our approach avoids these problems by strengthening network semantics. Table 1 summarizes the properties of these protocols.

3 Ordered Unreliable Multicast

We have argued for a separation of concerns between ordering and reliable delivery. Towards this end, we seek to design an *ordered* but *unreliable* network. In this section, we precisely define the properties that this network provides, and show how it can be realized efficiently using in-network processing.

We are not the first to argue for a network with ordered delivery semantics. Prior work has observed that some networks often deliver requests to replicas in the same order [50,58], that data center networks can be engineered to support a multicast primitive that has this property [53], and that it is possible to use this fact to design protocols that are more efficient in the common case [29, 36, 53]. We contribute by demonstrating that it is possible to build a network with ordering *guarantees* rather than probabilistic or best-effort properties. As we show in §5, doing so can support simpler and more efficient protocols.

Figure 1 shows the architecture of an OUM/NOPaxos deployment. All components reside in a single data center. OUM is implemented by components in the network along with a library, libOUM, that runs on senders and receivers. NOPaxos is a replication system that uses libOUM; clients use libOUM to send messages, and replicas use libOUM to receive clients' messages.

3.1 Ordered Unreliable Multicast Properties

We begin by describing the basic primitive provided by our networking layer: *ordered unreliable multicast*. More specifically, our model is an *asynchronous, unreliable network* that supports *ordered multicast* with *multicast drop detection*. These properties are defined as follows:

- **Asynchrony:** There is no bound on the latency of message delivery.

- **Unreliability:** The network does not guarantee that any message will ever be delivered to any recipient.

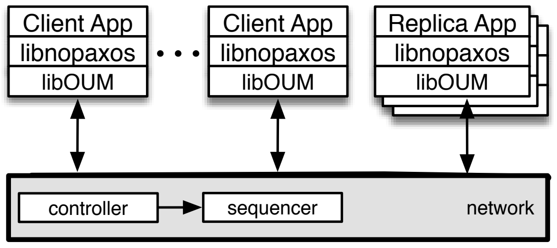


Figure 1: Architecture of NOPaxos.

- **Ordered Multicast**: The network supports a multicast operation such that if two messages, m and m', are multicast to a set of processes, R, then all processes in R that receive m and m' receive them in the same order.

- **Multicast Drop Detection**: If some message, m, is multicast to some set of processes, R, then either: (1) every process in R receives m or a notification that there was a dropped message before receiving the next multicast, or (2) no process in R receives m or a dropped message notification for m.[1]

The asynchrony and unreliability properties are standard in network design. Ordered multicast is not: existing multicast mechanisms do not exhibit this property, although Mostly-Ordered Multicast provides it on a best-effort basis [53]. Importantly, our model requires that any pair of multicast messages successfully sent to the same group are *always* delivered in the same order to all receivers – unless one of the messages is not received. In this case, however, the receiver is notified.

3.2 OUM Sessions and the libOUM API

Our OUM primitive is implemented using a combination of a network-layer sequencer and a communication library called libOUM. libOUM's API is a refinement of the OUM model described above. An OUM *group* is a set of receivers and is identified by an IP address. We explain group membership changes in §5.2.5.

libOUM introduces an additional concept, *sessions*. For each OUM group, there are one or more sessions, which are intervals during which the OUM guarantees

[1] This second case can be thought of as a *sender omission*, whereas the first case can be thought of as a *receiver omission*, with the added drop notification guarantee.

libOUM Sender Interface

- `send(addr destination, byte[] message)` — send a message to the given OUM group

libOUM Receiver Interface

- `getMessage()` — returns the next message, a DROP-NOTIFICATION, or a SESSION-TERMINATED error
- `listen(int sessionNum, int messageNum)` — resets libOUM to begin listening in OUM session *sessionNum* for message *messageNum*

Figure 2: The libOUM interface.

hold. Conceptually, the stream of messages being sent to a particular group is divided into consecutive OUM sessions. From the beginning of an OUM session to the time it terminates, all OUM guarantees apply. However, OUM sessions are not guaranteed to terminate at the same point in the message stream for each multicast receiver: an arbitrary number of messages at the end of an OUM session could be dropped without notification, and this number might differ for each multicast receiver. Thus, each multicast recipient receives a prefix of the messages assigned to each OUM session, where some messages are replaced with drop notifications.

Sessions are generally long-lived. However, rare, exceptional network events (sequencer failures) can terminate them. In this case, the application is notified of session termination and then must ensure that it is in a consistent state with the other receivers before listening for the next session. In this respect, OUM sessions resemble TCP connections: they guarantee ordering within their lifetime, but failures may cause them to end.

Applications access OUM sessions via the libOUM interface (Figure 2). The receiver interface provides a `getMessage()` function, which returns either a message or a DROP-NOTIFICATION during an OUM session. When an OUM session terminates, `getMessage()` returns a special value, SESSION-TERMINATED, until the user of libOUM starts the next OUM session. To begin listening to the next OUM session and receiving its messages and DROP-NOTIFICATIONs, the receiver calls `listen(int newSessionNum, 0)`. To start an OUM session at a particular position in the message stream, the receiver can call `listen(int sessionNum, int messageNum)`. Users of libOUM must ensure that all OUM receivers begin listening to the new session in a consistent state.

4 OUM Design and Implementation

We implement OUM in the context of a single data center network. The basic design is straightforward: the network routes all packets destined for a given OUM group through a single *sequencer*, a low-latency device that serves one purpose: to add a sequence number to each packet before forwarding it to its destination. Since all packets have been marked with a sequence number, the libOUM library can ensure ordering by discarding messages that are received out of order and detect and report dropped messages by noticing gaps in the sequence number.

Achieving this design poses three challenges. First, the network must serialize all requests through the sequencer; we use software-defined networking (SDN) to provide this *network serialization* (§4.1). Second, we must implement a sequencer capable of high throughput and low latency. We present three such implementations in §4.2: a zero-additional-latency implementation for programmable data center switches, a middlebox-like prototype using a network processor, and a pure-software implementation. Finally, the system must remain robust to failures of network components, including the sequencer (§4.3).

4.1 Network Serialization

The first aspect of our design is *network serialization*, where all OUM packets for a particular group are routed through a sequencer on the common path. Network serialization was previously used to implement a best-effort multicast [53]; we adapt that design here.

Our design targets a data center that uses software-defined networking, as is common today. Data center networks are engineered networks constructed with a particular topology – generally some variant of a multi-rooted tree. A traditional design calls for a three-level tree topology where many top-of-rack switches, each connecting to a few dozen server, are interconnected via aggregation switches that themselves connect through core switches. More sophisticated topologies, such as fat-tree or Clos networks [1,22,43,46] extend this basic design to support large numbers of physical machines using many commodity switches and often provide full bisection bandwidth. Figure 3 shows the testbed we use, implementing a fat-tree network [1].

Software-defined networking additionally allows the data center network to be managed by a central controller. This controller can install custom forwarding, filtering, and rewriting rules in switches. The current generation of SDN switches, e.g., OpenFlow [44], allow these rules to be installed at a per-flow granularity, matching on a fixed set of packet headers.

To implement network serialization, we assign each OUM group a distinct address in the data center network that senders can use to address messages to the group. The SDN controller installs forwarding rules for this address that route messages through the sequencer, then to group members.

To do this, the controller must select a sequencer for each group. In the most efficient design, switches themselves are used as sequencers (§4.2.1). In this case, the controller selects a switch that is a common ancestor of all destination nodes in the tree hierarchy to avoid increasing path lengths, e.g., a root switch or an aggregation switch if all receivers are in the same subtree. For load balancing, different OUM groups are assigned different sequencers, e.g., using hash-based partitioning.

Figure 3 shows an example of network serialization forwarding paths in a 12-switch, 3-level fat tree network. Sequencers are implemented as network processors (§4.2.2) connected to root switches. Messages from a client machine are first forwarded upward to the designated sequencer – here, attached to the leftmost root switch – then distributed downward to all recipients.

Network serialization could create longer paths than traditional IP multicast because all traffic must be routed to the sequencer, but this effect is minimal in practice. We quantified this latency penalty using packet-level network simulation. The simulated network contained 2,560 end-hosts and 119 switches configured in a 3-level fat tree network, with background traffic modeled on Microsoft data centers [4]. Each client sent multicast messages to a random group of 5 receivers. In 88% of cases, network serialization added *no additional latency* for the message to be received by a quorum of 3 receivers; the 99th-percentile was less than 5 μs of added latency. This minimal increase in latency is due to the fact that the sequencer is a least- common-ancestor switch of the replica group, and most packets have to traverse that switch anyway to reach a majority of the group.

4.2 Implementing the Sequencer

The sequencer plays a simple but critical role: assigning a sequence number to each message destined for a particular OUM group, and writing that sequence number into the packet header. This establishes a total order over packets and is the key element that elevates our design from a best-effort ordering property to an ordering guarantee. Even if packets are dropped (e.g., due to congestion or link failures) or reordered (e.g., due to multipath effects) in the network, receivers can use the sequence numbers to ensure that they process packets in order and deliver drop notifications for missing packets.

Sequencers maintain one counter per OUM group. For every packet destined for that group, they increment the counter and write it into a designated field in the packet header. The counter must be incremented by 1 on each packet (as opposed to a timestamp, which monotonically increases but may have gaps). This counter lets libOUM return DROP-NOTIFICATIONs when it notices gaps in the sequence numbers of incoming messages. Sequencers also maintain and write into each packet the OUM *session number* that is used to handle sequencer failures; we describe its use in §4.3.

Our sequencer design is general; we discuss three possible implementations here. The most efficient one targets upcoming programmable network switches, using the switch itself as the sequencer, incurring no latency cost (§4.2.1). As this hardware is not yet available, we describe a prototype that uses a network processor to implement a middlebox-like sequencer (§4.2.2). Finally, we discuss using an end-host as a sequencer (§4.2.3).

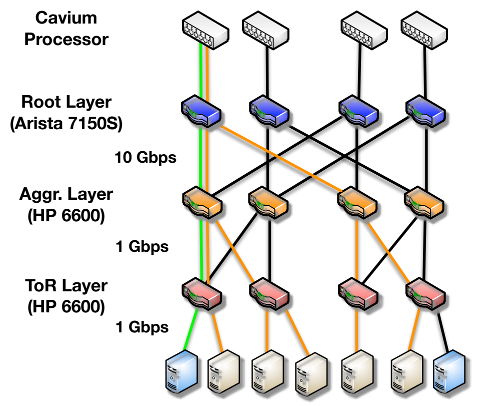


Figure 3: Testbed network topology. Green lines indicate the upward path from a client to the sequencer, and orange lines indicate the downward path from the sequencer to receivers.

4.2.1 In-Switch Sequencing

Ideally, switches themselves could serve as sequencers. The benefit of doing so is latency: packets could be sequenced by one of the switches through which they already transit, rather than having to be redirected to a dedicated device. Moreover, switching hardware is highly optimized for low-latency packet processing, unlike end-hosts.

Using a switch as a sequencer is made possible by the increasing ability of data center switches to perform flexible, per-packet computations. An emerging class of switch architectures – such as Reconfigurable Match Tables [10], Intel's FlexPipe [49], and Cavium's XPliant [59] – allow the switch's behavior to be controlled on a per-packet granularity, supporting the parsing and matching of arbitrary packet fields, rewriting of packet contents, and maintaining of small amounts of state between packets. Exposed through high-level languages like P4 [9], this increased flexibility lets us consider network switches as not simply forwarding elements, but as devices with computational ability.

We implemented our switch sequencing functionality in the P4 language, which allows it to be compiled and deployed to upcoming programmable switches as well as software switches. Our implementation uses the reconfigurable parser capabilities of these switches to define a custom packet header that includes the OUM sequence and session numbers. It uses stateful memory (register arrays) to store the current sequence number for every OUM group and increments it on each packet. Complete NOPaxos P4 code is available in [39].

Programmable switches capable of this processing are not yet commercially available, although we expect them to be within the next year. Therefore, we cannot evaluate their performance, but there is reason to believe they can execute this processing with no measurable increase in latency. As evidence, Intel's FlexPipe chips (now available, e.g., in the Arista 7150 switch) can modify packets to include the egress timestamp with zero latency cost [2, 49].

We note that a network switch provides orders-of-magnitude lower latency and greater reliability [21] than an end-host. Today's fastest cut-through switches can consistently process packets in approximately 300 ns [2], while a typical Linux server has median latency in the 10–100 μs range and 99.9th-percentile latency over 5 *ms* [40]. This trend seems unlikely to change: even with high-performance server operating systems [3,52], NIC latency remains an important factor [20]. At the same time, the limited computational model of the switch requires a careful partitioning of functionality between the network and application. The OUM model offers such a design.

4.2.2 Hardware Middlebox Prototype Sequencing

Because available switches do not provide the necessary flexibility to run P4 programs, we implemented a prototype using existing OpenFlow switches and a network processor.

This prototype is part of the testbed that we use to evaluate our OUM model and its uses for distributed protocols. This testbed simulates the 12-switch, 3-layer fat-tree network configuration depicted in Figure 3. We implemented it on three physical switches by using VLANs and appropriate OpenFlow forwarding rules to emulate separate virtual switches: two HP 6600 switches implement the ToR and aggregation tiers, and one Arista 7050S switch implements the core tier.

We implemented the sequencer as a form of middlebox using a Cavium Octeon II CN68XX network processor. This device contains 32 MIPS64 cores and supports 10 Gb/s Ethernet I/O. Users can customize network functionality by loading C binaries that match, route, drop or modify packets going through the processor. Onboard DRAM maintains per-group state. We attached the middlebox to the root switches and installed OpenFlow rules to redirect OUM packets to the middlebox.

This implementation does not provide latency as low as the switch-based sequencer; routing traffic through the network processor adds latency. We measured this latency to be 8 μs in the median case and 16 μs in the 99th percentile. This remains considerably lower than implementing packet processing in an end-host.

4.2.3 End-host Sequencing

Finally, we also implemented the sequencing functionality on a conventional server. While this incurs higher latency, it allows the OUM abstraction to be implemented without any specialized hardware. Nevertheless, using a dedicated host for network-level sequencing can still provide throughput, if not latency, benefits as we demonstrate in §6. We implemented a simple Linux program that uses raw sockets to access packet headers.

4.2.4 Sequencer Scalability

Since all OUM packets for a particular group go through the sequencer, a valid concern is whether the sequencer will become the performance bottleneck. Switches and network processors are designed to process packets at line rate and thus will not become the bottleneck for a single OUM group (group receivers are already limited by the link bandwidth). Previous work [28] has demonstrated that an end-host sequencer using RDMA can process close to 100 million requests per second, many more than any single OUM group can process. We note that different OUM groups need not share a sequencer, and therefore deployment of multiple OUM groups can scale horizontally.

4.3 Fault Tolerance

Designating a sequencer and placing it on the common path for all messages to a particular group introduces an obvious challenge: what if it fails or becomes unreachable? If link failures or failures of other switches render the sequencer unreachable, local rerouting mechanisms may be able to identify an alternate path [43]. However, if the sequencer itself fails, or local rerouting is not possible, replacing the sequencer becomes necessary.

In our design, the network controller monitors the sequencer's availability. If it fails or no longer has a path to all OUM group members, the controller selects a different switch. It reconfigures the network to use this new sequencer by updating routes in other switches. During the reconfiguration period, multicast messages may not be delivered. However, failures of root switches happen infrequently [21], and rerouting can be completed within a few milliseconds [43], so this should not significantly affect system availability.

We must also ensure that the ordering guarantee of multicast messages is robust to sequencer failures. This requires the continuous, ordered assignment of sequence numbers even when the network controller fails over to a new sequencer.

To address this, we introduce a unique, monotonically increasing *session number*, incremented each time sequencer failover occurs. When the controller detects a sequencer failure, it updates the forwarding rules and contacts the new sequencer to set its local session number to the appropriate value. As a result, the total order of messages follows the lexicographical order of the $\langle$*session-number*, *sequence-number*$\rangle$ tuple, and clients can still discard packets received out of order.

Once libOUM receives a message with a session number higher than the receiver is listening for, it realizes that a new sequencer is active and stops delivering messages from the old session. However, libOUM does not know if it missed any packets from the old sequencer. As a result, it cannot deliver DROP-NOTIFICATIONs during a session change. Instead, it delivers a SESSION-TERMINATED notification, exposing this uncertainty to the application. NOPaxos, for example, resolves this by executing a view change (§5.2.3) so that replicas agree on exactly which requests were received in the old session.

The network controller must ensure that session numbers for any given group monotonically increase, even across controller failures. Many design options are available, for example using timestamps as session numbers, or recording session numbers in stable or replicated storage. Our implementation uses a Paxos-replicated controller group, since SDN controller replication is already common in practice [26, 31]. We note that our replication protocol, NOPaxos (§5), is completely decoupled from controller replication, and the controller updates only on sequencer failures, not for every NOPaxos request.

5 NOPaxos

NOPaxos, or Network-Ordered Paxos, is a new replication protocol which leverages the Ordered Unreliable Multicast sessions provided by the network layer.

5.1 Model

NOPaxos replicas communicate over an asynchronous network that provides OUM sessions (via libOUM). NOPaxos requires the network to provide ordered but unreliable delivery of multicast messages within a session. In the normal case, these messages are delivered sequentially and are not dropped; however, it remains robust to dropped packets (presented as DROP-NOTIFICATION through libOUM). NOPaxos is also robust to SESSION-TERMINATED notifications that occur if the sequencer fails. These network anomalies do not affect NOPaxos's safety guarantees, and we discuss how they affect NOPaxos's performance in §6.

NOPaxos assumes a crash failure model. It uses $2f+1$ replicas, where f replicas are allowed to fail. However, in the presence of more than f failures, the system still guarantees safety. Furthermore, NOPaxos guarantees safety even in an asynchronous network with no bound on message latency (provided the OUM guarantees continue to hold).

NOPaxos provides linearizability of client requests. It provides at-most-once semantics using the standard mechanism of maintaining a table of the most recent request from each client [42].

5.2 Protocol

Overview. NOPaxos is built on top of the guarantees of the OUM network primitive. During a single OUM session, REQUESTs broadcast to the replicas are totally ordered but can be dropped. As a result, the replicas have to agree only on which REQUESTs to execute and which to permanently ignore, a simpler task than agreeing on the order of requests. Conceptually, this is equivalent to running multiple rounds of *binary consensus*. However, NOPaxos must explicitly run this consensus only when DROP-NOTIFICATIONs are received. To switch OUM sessions (in the case of sequencer failure), the replicas must agree on the contents of their shared log before they start listening to the new session.

To these ends, NOPaxos uses a view-based approach: each view has a single OUM *session-num* and a single replica acting as *leader*. The leader executes requests and drives the agreement to skip a dropped request. That is, it decides which of the sequencer's REQUESTs to ignore and treat as NO-OPs. The view ID is a tuple $\langle$*leader-num*, *session-num*$\rangle$. Here, *leader-num* is incremented each time a new leader is needed; the current leader of any view is *leader-num* $\pmod n$; and *session-num* is the latest session ID from libOUM. View IDs in NOPaxos are partially ordered.[2] However, the IDs of all views that successfully start will be comparable.

In the normal case, the replicas receive a REQUEST from libOUM. The replicas then reply to the client, the leader replying with the result of the REQUEST, so the client's REQUEST is processed in only a single round-trip. NOPaxos uses a single round-trip in the normal case because, like many speculative protocols, the client checks the durability of requests. However, unlike most speculative protocols, NOPaxos clients have a guarantee regarding ordering of operations; they need only check that the operation was received.

When replicas receive a DROP-NOTIFICATION from libOUM, they first try to recover the missing REQUEST from each other. Failing that, the leader initiates a round of agreement to commit a NO-OP into the corresponding slot in the log. Finally, NOPaxos uses a view change protocol to handle leader failures and OUM session termination while maintaining consistency.

Outline. NOPaxos consists of four subprotocols:

- *Normal Operations* (§5.2.1): NOPaxos processes client REQUESTs in a single round-trip in the normal case.
- *Gap Agreement* (§5.2.2): NOPaxos ensures correctness in the face of DROP-NOTIFICATIONs by having the

[2] That is, $v_1 \le v_2$ iff both v_1's *leader-num* and *session-num* are less than or equal to v_2's.

Replica:

- *replica-num* — replica number
- *status* — one of `Normal` or `ViewChange`
- *view-id* = ⟨*leader-num*, *session-num*⟩ — the view number, a tuple of the current leader number and OUM session number, partially ordered, initially ⟨0, 0⟩
- *session-msg-num* — the number of messages (REQUESTs or DROP-NOTIFICATIONs) received in this OUM session
- *log* — client REQUESTs and NO-OPs in sequential order
- *sync-point* — the latest synchronization point

Figure 4: Local state of NOPaxos replicas.

replicas reach agreement on which sequence numbers should be permanently dropped.

- *View Change* (§5.2.3): NOPaxos ensures correctness in the face of leader failures or OUM session termination using a variation of a standard view change protocol.

- *Synchronization* (§5.2.4): Periodically, the leader synchronizes the logs of all replicas.

Figure 4 illustrates the state maintained at each NOPaxos replica. Replicas tag all messages sent to each other with their current *view-id*, and while in the Normal Operations, Gap Agreement, and Synchronization subprotocols, *replicas ignore all messages from different views*. Only in the View Change protocol do replicas with different *view-id*s communicate.

5.2.1 Normal Operations

In the normal case when replicas receive REQUESTs instead of DROP-NOTIFICATIONs, client requests are committed and executed in a single phase. Clients broadcast ⟨REQUEST, *op*, *request-id*⟩ to all replicas through libOUM, where *op* is the operation they want to execute, and *request-id* is a unique id used to match requests and their responses.

When each replica receives the client's REQUEST, it increments *session-msg-num* and appends *op* to the log. If the replica is the leader of the current view, it executes the *op* (or looks up the previous result if it is a duplicate of a completed request). Each replica then replies to the client with ⟨REPLY, *view-id*, *log-slot-num*, *request-id*, *result*⟩, where *log-slot-num* is the index of *op* in the log. If the replica is the leader, it includes the *result* of the operation; NULL otherwise.

The client waits for REPLYs to the REQUEST with matching *view-id*s and *log-slot-num*s from $f+1$ replicas, where one of those replicas is the leader of the view. This indicates that the request will remain persistent even across view changes. If the client does not receive the required REPLYs within a timeout, it retries the request.

5.2.2 Gap Agreement

NOPaxos replicas always process operations in order. When a replica receives a DROP-NOTIFICATION from libOUM (and increments its *session-msg-num*), it must either recover the contents of the missing request or prevent it from succeeding before moving on to subsequent requests. Non-leader replicas do this by contacting the leader for a copy of the request. If the leader itself receives a DROP-NOTIFICATION, it coordinates to commit a NO-OP operation in place of that request:

1. If the leader receives a DROP-NOTIFICATION, it inserts a NO-OP into its *log* and sends a ⟨GAP-COMMIT, *log-slot*⟩ to the other replicas, where *log-slot* is the slot into which the NO-OP was inserted.

2. When a non-leader replica receives the GAP-COMMIT and has filled all log slots up to the one specified by the leader, it inserts a NO-OP into its *log* at the specified location[3] (possibly overwriting a REQUEST) and replies to the leader with a ⟨GAP-COMMIT-REP, *log-slot*⟩.

3. The leader waits for f GAP-COMMIT-REPs (retrying if necessary).

Clients need not be notified explicitly when a NO-OP has been committed in place of one of their requests. They simply retry their request after failing to receive a quorum of responses. Note that the retried operation will be considered a new request and will have a new slot in the replicas' logs. Replicas identify duplicate client requests by checking if they have processed another request with the same *client-id* and *request-id*, as is commonly done in other protocols.

This protocol ensures correctness because clients do not consider an operation completed until they receive a response from the leader, so the leader can propose a NO-OP regardless of whether the other replicas received the REQUEST. However, before proceeding to the next sequence number, the leader must ensure that a majority of replicas have learned of its decision to commit a NO-OP. When combined with the view change protocol, this ensures that the decision persists even if the leader fails.

As an optimization, the leader can first try to contact the other replicas to obtain a copy of the REQUEST and initiate the gap commit protocol only if no replicas respond before a timeout. While not necessary for correctness, this reduces the number of NO-OPs.

[3] If the replica had not already filled *log-slot* in its log or received a DROP-NOTIFICATION for that slot when it inserted the NO-OP, it ignores the next REQUEST or DROP-NOTIFICATION from libOUM (and increments *session-msg-num*), maintaining consistency between its position in the OUM session and its log.

5.2.3 View Change

During each view, a NOPaxos group has a particular leader and OUM session number. NOPaxos must perform view changes to ensure progress in two cases: (1) when the leader is suspected of having failed (e.g, by failing to respond to pings), or (2) when a replica detects the end of an OUM session. To successfully replace the leader or move to a new OUM session, NOPaxos runs a view change protocol. This protocol ensures that all successful operations from the old view are carried over into the new view and that all replicas start the new view in a consistent state.

NOPaxos's view change protocol resembles that used in Viewstamped Replication [42]. The principal difference is that NOPaxos views serve two purposes, and so NOPaxos view IDs are therefore a tuple of ⟨*leader-num*, *session-num*⟩ rather than a simple integer. A view change can increment either one. However, NOPaxos ensures that each replica's *leader-num* and *session-num* never go backwards. This maintains a total order over all views that successfully start.

1. A replica initiates a view change when: (1) it suspects that the leader in its current view has failed; (2) it receives a SESSION-TERMINATED notification from libOUM; or (3) it receives a VIEW-CHANGE or VIEW-CHANGE-REQ message from another replica with a higher *leader-num* or *session-num*. In all cases, the replica appropriately increments the *leader-num* and/or *session-num* in its *view-id* and sets its *status* to `ViewChange`. If the replica incremented its *session-num*, it resets its *session-msg-num* to 0.

   It then sends ⟨VIEW-CHANGE-REQ, *view-id*⟩ to the other replicas and ⟨VIEW-CHANGE, *view-id*, v', *session-msg-num*, *log*⟩ to the leader of the new view, where v' is the view ID of the last view in which its *status* was `Normal`. While in `ViewChange` status, the replica ignores all replica-to-replica messages (except START-VIEW, VIEW-CHANGE, and VIEW-CHANGE-REQ).

   If the replica ever times out waiting for the view change to complete, it simply rebroadcasts the VIEW-CHANGE and VIEW-CHANGE-REQ messages.

2. When the leader for the new view receives $f+1$ VIEW-CHANGE messages (including one from itself) with matching *view-id*s, it performs the following steps:

   - The leader merges the *log*s *from the most recent (largest) view* in which the replicas had *status* `Normal`.[4] For each slot in the log, the merged result is a NO-OP if any log has a NO-OP. Otherwise, the result is a REQUEST if at least one has a REQUEST. It then updates its *log* to the merged one.
   - The leader sets its *view-id* to the one from the VIEW-CHANGE messages and its *session-msg-num* to the highest out of all the messages used to form the merged log.
   - It then sends ⟨START-VIEW, *view-id*, *session-msg-num*, *log*⟩ to all replicas (including itself).

3. When a replica receives a START-VIEW message with a *view-id* greater than or equal to its current *view-id*, it first updates its *view-id*, *log*, and *session-msg-num* to the new values. It then calls `listen(session-num, session-msg-num)` in libOUM. The replica sends REPLYs to clients for all new REQUESTs added to its log (executing them if the replica is the new leader). Finally, the replica sets its *status* to `Normal` and begins receiving messages from libOUM again.[5]

5.2.4 Synchronization

During any view, only the leader executes operations and provides results. Thus, all successful client REQUESTs are committed on a *stable log* at the leader, which contains only persistent client REQUESTs. In contrast, non-leader replicas might have *speculative* operations throughout their logs. If the leader crashes, the view change protocol ensures that the new leader first recreates the stable log of successful operations. However, it must then execute all operations before it can process new ones. While this protocol is correct, it is clearly inefficient.

Therefore, as an optimization, NOPaxos periodically executes a synchronization protocol in the background. This protocol ensures that all other replicas learn which operations have successfully completed and which the leader has replaced with NO-OPs. That is, synchronization ensures that all replicas' logs are stable up to their *sync-point* and that they can safely execute all REQUESTs up to this point in the background.

For brevity, we omit the details of this protocol. See [39] for the full specification.

5.2.5 Recovery and Reconfiguration

While the NOPaxos protocol as presented above assumes a crash failure model and a fixed replica group, it can

[4] While *view-id*s are only partially ordered, because individual replicas' *view-id*s only increase and views require a quorum of replicas to start, all views that successfully start *are* comparable – so identifying the view with the highest number is in fact meaningful. For a full proof of this fact, see [39].

[5] Replicas also send an acknowledgment to the leader's START-VIEW message, and the leader periodically resends the START-VIEW to those replicas from whom it has yet to receive an acknowledgment.

also facilitate recovery and reconfiguration using adaptations of standard mechanisms (e.g. Viewstamped Replication [42]). While the recovery mechanism is a direct equivalent of the Viewstamped Replication protocol, the reconfiguration protocol additionally requires a membership change in the OUM group. The OUM membership is changed by contacting the controller and having it install new forwarding rules for the new members, as well as a new *session-num* in the sequencer (terminating the old session). The protocol then ensures all members of the new configuration start in a consistent state.

5.3 Benefits of NOPaxos

NOPaxos achieves the theoretical minimum latency and maximum throughput: it can execute operations in *one round-trip* from the client to the replicas and does not require replicas to coordinate on each request. By relying on the network to stamp requests with sequence numbers, it requires replies only from a *simple majority* of replicas and uses a *cheaper* and *rollback-free* mechanism to correctly account for network anomalies.

The OUM session guarantees mean that the replicas already agree on the ordering of all operations. As a consequence, clients need not wait for a superquorum of replicas to reply, as in Fast Paxos and Speculative Paxos (and as is required by any protocol that provides fewer message delays than Paxos in an asynchronous, unordered network [37]). In NOPaxos, a simple majority of replicas suffices to guarantee the durability of a REQUEST in the replicas' shared log.

Additionally, the OUM guarantees enable NOPaxos to avoid expensive mechanisms needed to detect when replicas are not in the same state, such as using hashing to detect conflicting logs from replicas. To keep the replicas' logs consistent, the leader need only coordinate with the other replicas when it receives DROP-NOTIFICATIONs. Committing a NO-OP takes but a single round-trip and requires no expensive reconciliation protocol.

NOPaxos also avoids rollback, which is usually necessary in speculative protocols. It does so not by coordinating on every operation, as in non-speculative protocols, but by having only the leader execute operations. Non-leader replicas do not execute requests during normal operations (except, as an optimization, when the synchronization protocol indicates it is safe to do so), so they need not rollback. The leader executes operations speculatively, without coordinating with the other replicas, but clients do not accept a leader's response unless it is supported by matching responses from f other replicas. The only rare case when a replica will execute an operation that is not eventually committed is if a functioning leader is incorrectly replaced through a view change, losing some operations it executed. Because this case is rare, it is reasonable to handle it by having the ousted leader transfer application state from another replica, rather than application-level rollback.

Finally, unlike many replication protocols, NOPaxos replicas send and receive a constant number of messages for each REQUEST in the normal case, irrespective of the total number of replicas. This means that NOPaxos can be deployed with an increasing number of replicas without the typical performance degradation, allowing for greater fault-tolerance. §6.3 demonstrates that NOPaxos achieves the same throughput regardless of the number of replicas.

5.4 Correctness

NOPaxos guarantees *linearizability*: that operations submitted by multiple concurrent clients appear to be executed by a single, correct machine. In a sense, correctness in NOPaxos is a much simpler property than in other systems, such as Paxos and Viewstamped Replication [33, 48], because the replicas need not agree on the order of the REQUESTs they execute. Since the REQUEST order is already provided by the guarantees of OUM sessions, the replicas must only agree on *which* REQUESTs to execute and which REQUESTs to drop.

Below, we sketch the proof of correctness for the NOPaxos protocol. For a full, detailed proof, see [39]. Additionally, see [39] for a TLA+ specification of the NOPaxos protocol.

Definitions. We say that a REQUEST or NO-OP is *committed* in a log slot if it is processed by $f+1$ replicas with matching *view-id*s, including the leader of that view. We say that a REQUEST is *successful* if it is committed and the client receives the $f+1$ suitable REPLYs. We say a log is *stable* in view v if it will be a prefix of the log of every replica in views higher than v.

Sketch of Proof. During a view, a leader's log grows monotonically (i.e., entries are only appended and never overwritten). Also, leaders execute only the first of duplicate REQUESTs. Therefore, to prove linearizability it is sufficient to show that: (1) every successful operation was appended to a stable log at the leader and that the resulting log is also stable, and (2) replicas always start a view listening to the correct *session-msg-num* in an OUM session (i.e., the message corresponding to the number of REQUESTs or NO-OPs committed in that OUM session).

First, note that any REQUEST or NO-OP that is committed in a log slot will stay in that log slot for all future views: it takes $f+1$ replicas to commit a view and $f+1$ replicas to complete a view change, so, by quorum intersection, at least one replica initiating the view change will have received the REQUEST or NO-OP. Also, because it takes the leader to commit a REQUEST or NO-OP and its log grows monotonically, only a single REQUEST or NO-OP is ever committed in the same slot during a view. Therefore, any log consisting of only committed REQUESTs and

NO-OPs is stable.

Next, every view that starts (i.e., $f+1$ replicas receive the START-VIEW and enter `Normal` status) trivially starts with a log containing only committed REQUESTs and NO-OPs. Replicas send REPLYs to a REQUEST only after all log slots before the REQUEST's slot have been filled with REQUESTs or NO-OPs; further, a replica inserts a NO-OP only if the leader already inserted that NO-OP. Therefore, if a REQUEST is committed, all previous REQUESTs and NO-OPs in the leader's log were already committed.

This means that any REQUEST that is successful in a view must have been appended to a stable log at the leader, and the resulting log must also be stable, showing (1). To see that (2) is true, notice that the last entry in the combined *log* formed during a view change and the *session-msg-num* are taken from the same replica(s) and therefore must be consistent.

NOPaxos also guarantees *liveness* given a sufficient amount of time during which the following properties hold: the network over which the replicas communicate is fair-lossy; there is some bound on the relative processing speeds of replicas; there is a quorum of replicas that stays up; there is a replica that stays up that no replica suspects of having failed; all replicas correctly suspect crashed nodes of having failed; no replica receives a DROP-NOTIFICATION or SESSION-TERMINATED from libOUM; and clients' REQUESTs eventually get delivered through libOUM.

6 Evaluation

We implemented the NOPaxos protocol in approximately 5,000 lines of C++ code. We ran our experiments using the 3-level fat-tree network testbed shown in Figure 3. All clients and replicas ran on servers with 2.5 GHz Intel Xeon E5-2680 processors and 64GB of RAM. All experiments used five replicas (thereby tolerating two replica failures).

To evaluate the performance of NOPaxos, we compared it to four other replication protocols: Paxos, Fast Paxos, Paxos with batching, and Speculative Paxos; we also evaluated it against an unreplicated system that provides no fault tolerance. Like NOPaxos, the clients in both Speculative Paxos and Fast Paxos multicast their requests to the replicas through a root serialization switch to minimize message reordering. Requests from NOPaxos clients, however, are also routed through the Cavium processor to be stamped with the sequencer's OUM session number and current request sequence number. For the batching variant of Paxos, we used a sliding-window technique where the system adaptively adjusts the batch size, keeping at least one batch in progress at all times; this approach reduces latency at low load while still providing throughput benefits at high load [13].

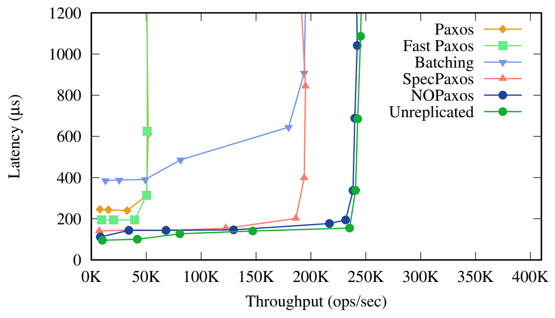


Figure 5: Latency vs. throughput comparison for testbed deployment of NOPaxos and other protocols.

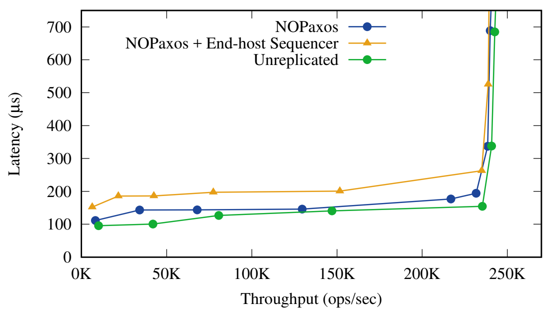


Figure 6: Comparison of running NOPaxos with the prototype Cavium sequencer and an end-host sequencer.

6.1 Latency vs. Throughput

To compare the latency and throughput of NOPaxos and the other four protocols, we ran each system with an increasing number of concurrent closed-loop clients. Figure 5 shows results of this experiment. NOPaxos achieves a much higher maximum throughput than Paxos and Fast Paxos (370% increases in both cases) without any additional latency. The leaders in both Paxos and Fast Paxos send and receive more messages than the other replicas, and the leaders' message processing quickly becomes the bottleneck of these systems. NOPaxos has no such inefficiency. NOPaxos also achieves higher throughput than Speculative Paxos (24% increase) because Speculative Paxos requires replicas to compute hashes of their logs for each client request.

Figure 5 also shows that NOPaxos has lower latency (111 μs) than Paxos (240 μs) and Fast Paxos (193 μs) because NOPaxos requires fewer message delays in the normal case. Speculative Paxos also has higher latency than NOPaxos because clients must wait for a superquorum of replica replies instead of NOPaxos's simple quorum.

Batching improves Paxos's throughput by reducing the number of messages sent by the leader. Paxos with batching is able to reach a maximum throughput equivalent to Speculative Paxos. However, batching also increases the latency of Paxos (385 μs at low load and 907 μs at maximum throughput). NOPaxos attains *both* higher throughput and lower latency than Paxos with batching.

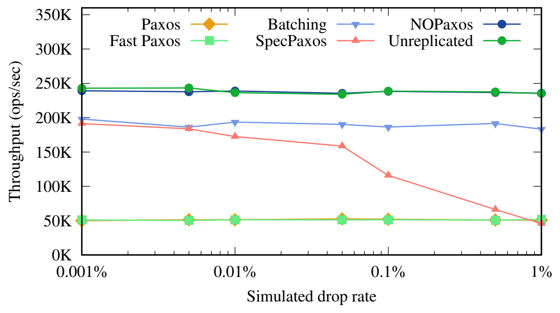


Figure 7: Maximum throughput with simulated packet dropping.

NOPaxos is able to attain throughput within 2% of an unreplicated system and latency within 16 μs. However, we note that our middlebox prototype adds around 8 μs to NOPaxos's latency. We envision that implementing the sequencer in a switch could bring NOPaxos's latency even closer to the unreplicated system. This demonstrates that NOPaxos can achieve close to optimal performance while providing fault-tolerance and strong consistency.

We also evaluated the performance of NOPaxos when using an end-host as the sequencer instead of the network processor. Figure 6 shows that NOPaxos still achieves impressive throughput when using an end-host sequencer, though at a cost of 36% more latency due to the additional message delay required.

6.2 Resilience to Network Anomalies

To test the performance of NOPaxos in an unreliable network, we randomly dropped a fraction of all packets. Figure 7 shows the maximum throughput of the five protocols and the unreplicated system with an increasing packet drop rate. Paxos's and Fast Paxos's throughput do not decrease significantly, while Paxos with batching shows a larger drop in throughput due to frequent state transfers. However, the throughput of Speculative Paxos drops substantially after 0.5% packet dropping, demonstrating NOPaxos's largest advantage over Speculative Paxos. When 1% of packets are dropped, Speculative Paxos's maximum throughput falls to that of Paxos. As discussed in §5.3, Speculative Paxos performs an expensive reconciliation protocol when messages are dropped and replica states diverge. NOPaxos is much more resilient to packet drops and reorderings. It achieves higher throughput than Paxos with batching and much higher throughput than Speculative Paxos at high drop rates. Even with a 1% message drop rate, NOPaxos's throughput does not drop significantly. Indeed, NOPaxos maintains throughput roughly equivalent to an unreplicated system, demonstrating its strong resilience to network anomalies.

6.3 Scalability

To test NOPaxos's scalability, we measured the maximum throughput of the five protocols running on increasing number of replicas. Figure 8 shows that both Paxos and

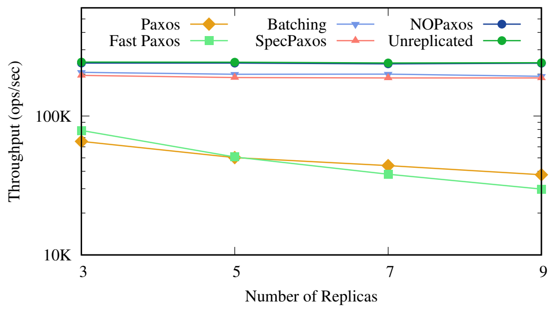


Figure 8: Maximum throughput with increasing number of replicas.

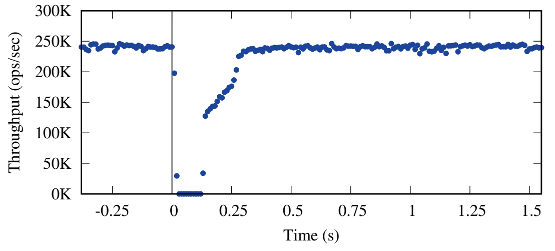


Figure 9: NOPaxos throughput during a sequencer failover.

Fast Paxos suffer throughput degradation proportional to the number of replicas because the leaders in those protocols have to process more messages from the additional replicas. Replicas in NOPaxos and Speculative Paxos, however, process a constant number of messages, so those protocols maintain their throughput when more replicas are added.

6.4 Sequencer Failover

NOPaxos relies on the sequencer to order client requests. We measured the throughput of NOPaxos during a sequencer failover (Figure 9). We ran NOPaxos at peak throughput for approximately 7 seconds. We then simulated a sequencer failure by sending the controller a notification message. The controller modified the routing rules in the network and installed a new session number in the sequencer (as described in §3). The throughput of the system drops to zero during the failover and takes approximately 110 ms to resume normal operations and approximately 270 ms to resume processing operations at peak throughput. Most of this delay is caused by the route update rather than the NOPaxos view change.

6.5 Application Performance

To further demonstrate the benefits of the NOPaxos protocol, we evaluated the performance of a distributed, in-memory key-value store. The key-value store uses two-phase commit and optimistic concurrency control to support serializable transactions, and each shard runs atop our replication framework. Clients issue `GET` and `PUT` requests within transactions. We benchmarked the key-value store using a workload based on the Retwis Twitter clone [38].

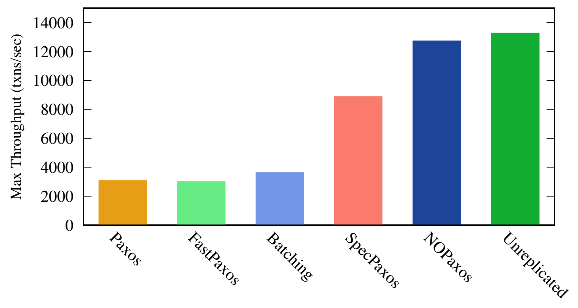


Figure 10: Maximum throughput achieved by a replicated transactional key-value store within 10 ms SLO.

Figure 10 shows the maximum throughput of the key-value store with a 10ms SLO. NOPaxos outperforms all other variants on this metric: it attains more than 4 times the throughput of Paxos, and outperforms the best prior protocol, Speculative Paxos, by 45%. Its throughput is also within 4% that of an unreplicated system.

7 Related Work

Our work draws on techniques from consensus protocol design as well as network-level processing mechanisms.

Consensus protocols Many protocols have been proposed for the equivalent problems of consensus, state machine replication, and atomic broadcast. Most closely related is a line of work on achieving better performance when requests *typically* arrive at replicas in the same order, including Fast Paxos [36], Speculative Paxos [53], and Optimistic Atomic Broadcast [29, 50, 51]; Zyzzyva [32] applies a similar idea in the context of Byzantine fault tolerant replication. These protocols can reduce consensus latency in a manner similar to NOPaxos. However, because requests are not *guaranteed* to arrive in the same order, they incur extra complexity and require supermajority quorum sizes to complete a request (either $2/3$ or $3/4$ of replicas rather than a simple majority). This difference is fundamental: the possibility of conflicting orders requires either an extra message round or a larger quorum size [37].

Another line of work aims to reduce latency and improve throughput by avoiding coordination for operations that are commutative or otherwise need not be ordered [12, 35, 45, 60]; this requires application support to identify commutative operations. NOPaxos avoids coordination for *all* operations.

Ordered Unreliable Multicast is related to a long line of work on totally ordered broadcast primitives, usually in the context of group communication systems [6, 7]. Years ago, a great debate raged in the SOSP community about the effectiveness and limits of this causal and totally ordered communication support (CATOCS) [5, 15]. Our work draws inspiration from both sides of this debate, but occupies a new point in the design space by splitting the responsibility between an ordered but unreliable communications layer and an application-level reliability layer. In particular, the choice to leave reliability to the application is inspired by the end-to-end argument [15, 55].

Network-level processing NOPaxos takes advantage of flexible network processing to implement the OUM model. Many designs have been proposed for flexible processing, including fully flexible, software-based designs like Click [30] and others based on network processors [57] or FPGA platforms [47]. At the other extreme, existing software defined networking mechanisms like OpenFlow [44] can easily achieve line-rate performance in commodity hardware implementations but lack the flexibility to implement our multicast primitive. We use the P4 language [9], which supports several high-performance hardware designs like Reconfigurable Match Tables [10].

These processing elements have generally been used for classic networking tasks like congestion control or queue management. A notable exception is SwitchKV [41], which uses OpenFlow switches for content-based routing and load balancing in key-value stores.

...and their intersection Recent work on Speculative Paxos and Mostly-Ordered Multicast proposes co-designing network primitives and consensus protocols to achieve faster performance. Our work takes the next step in this direction. While Speculative Paxos assumes only a best-effort ordering property, NOPaxos requires an ordering guarantee. Achieving this guarantee requires more sophisticated network support made possible with a programmable data plane (Speculative Paxos's Mostly-Ordered Multicast requires only OpenFlow support). However, as discussed in §5.3, NOPaxos achieves a simpler and more robust protocol as a result, avoiding the need for superquorums and speculation.

A concurrent effort, NetPaxos [18], also explores ways to use the network layer to improve the performance of a replication protocol. That work proposes moving the Paxos logic into switches, with one switch serving as a coordinator and others as Paxos acceptors. This logic can also be implemented using P4 [17]. However, as the authors note, this approach requires the switches to implement substantial parts of the logic, including storing potentially large amounts of state (the results of each consensus instance). Our work takes a more practical approach by splitting the responsibility between the OUM network model, which can be readily implemented, and the NOPaxos consensus protocol.

Other related work uses hardware acceleration to speed communication between nodes in a distributed system. FaRM [19] uses RDMA to bypass the kernel and minimize CPU involvement in remote memory accesses. Consensus in a Box [25] implements a standard atomic broadcast protocol entirely on FPGAs. NOPaxos provides more flexible deployment options. However, its protocol could be integrated with RDMA or other kernel-bypass networking for faster replica performance.

8 Conclusions

We presented a new approach to high-performance, fault-tolerant replication, one based on dividing the responsibility for consistency between the network layer and the replication protocol. In our approach, the network is responsible for ordering, while the replication protocol ensures reliable delivery. "Splitting the atom" in this way yields dramatic performance gains: network-level ordering, while readily achievable, supports NOPaxos, a simpler replication protocol that avoids coordination in most cases. The resulting system outperforms state-of-the-art replication protocols on latency, throughput, and application-level metrics, demonstrating the power of this approach. More significantly, it achieves both throughput and latency equivalent to an unreplicated system, proving that replication does not have to come with a performance cost.

Acknowledgments

We thank Irene Zhang, the anonymous reviewers, and our shepherd Dawn Song for their helpful feedback. This research was supported by the National Science Foundation under awards CNS-1518702 and CNS-1615102 and by gifts from Google and VMware.

References

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

[2] Arista Networks. 7150 series ultra low latency switch. `https://www.arista.com/assets/data/pdf/Datasheets/7150S_Datasheet.pdf`.

[3] A. Belay, G. Prekas, A. Klimovic, S. Grossman, C. Kozyrakis, and E. Bugnion. IX: A protected dataplane operating system for high throughput and low latency. In *Proceedings of the 11th USENIX Symposium on Operating Systems Design and Implementation (OSDI '14)*, Broomfield, CO, USA, Oct. 2014. USENIX.

[4] 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, Melbourne, Australia, 2010. ACM.

[5] K. Birman. A response to Cheriton and Skeen's criticism of causal and totally ordered communication. *ACM SIGOPS Operating Systems Review*, 28(1), Jan. 1994.

[6] 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.

[7] K. P. Birman and T. A. Joseph. Reliable communication in the presence of failures. *ACM Trans. Comput. Syst.*, 5(1), Jan. 1987.

[8] W. J. Bolosky, D. Bradshaw, R. B. Haagens, N. P. Kusters, and P. Li. Paxos replicated state machines as the basis of a high-performance data store. In *Proceedings of the 8th USENIX Symposium on Networked Systems Design and Implementation (NSDI '11)*, Boston, MA, USA, Apr. 2011. USENIX.

[9] 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), July 2014.

[10] 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 *Proceedings of ACM SIGCOMM 2013*. ACM, 2013.

[11] 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.

[12] L. Camargos, R. Schmidt, and F. Pedone. Multi-coordinated Paxos. Technical report, University of Lugano Faculty of Informatics, 2007/02, Jan. 2007.

[13] 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.

[14] T. D. Chandra, V. Hadzilacos, and S. Toueg. The weakest failure detector for solving consensus. *J. ACM*, 43(4), July 1996.

[15] D. R. Cheriton and D. Skeen. Understanding the limitations of causally and totally ordered communication. In *Proceedings of the 13th ACM Symposium on Operating Systems Principles (SOSP '93)*, Asheville, NC, USA, Dec. 1993. ACM.

[16] 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.

[17] H. T. Dang, P. Bressana, H. Wang, K. S. Lee, H. Weatherspoon, M. Canini, F. Pedone, and R. Soulé. Network hardware-accelerated consensus. Technical Report USI-INF-TR-2016-03, Università della Svizzera italiana, May 2016.

[18] H. T. Dang, D. Sciascia, M. Canini, F. Pedone, and R. Soulé. NetPaxos: Consensus at network speed. In *Proceedings of the 1st ACM SIGCOMM Symposium on Software Defined Networking Research*, SOSR '15, New York, NY, USA, 2015. ACM.

[19] A. Dragojević, D. Narayanan, M. Castro, and O. Hodson. FaRM: Fast remote memory. In *11th USENIX Symposium on Networked Systems Design and Implementation (NSDI 14)*, Seattle, WA, Apr. 2014. USENIX Association.

[20] M. Flajslik and M. Rosenblum. Network interface design for low latency request-response protocols. In *Proceedings of the 2013 USENIX Annual Technical Conference*, San Jose, CA, USA, June 2013. USENIX.

[21] P. Gill, N. Jain, and N. Nagappan. Understanding network failures in data centers: Measurement, anal-

ysis, and implications. In *Proceedings of ACM SIGCOMM 2011*, Toronto, ON, Canada, Aug. 2011.

[22] 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.

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

[24] 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.

[25] Z. István, D. Sidler, G. Alonso, and M. Vukolic. Consensus in a box: Inexpensive coordination in hardware. In *13th USENIX Symposium on Networked Systems Design and Implementation (NSDI 16)*, Santa Clara, CA, Mar. 2016. USENIX Association.

[26] S. Jain, A. Kumar, S. Mandal, J. Ong, L. Poutievski, A. Singh, S. Venkata, J. Wanderer, J. Zhou, M. Zhu, J. Zolla, U. Hölzle, S. Stuart, and A. Vahdat. B4: Experience with a globally-deployed software defined WAN. In *Proceedings of ACM SIGCOMM 2013*, Hong Kong, China, Aug. 2013. ACM.

[27] F. P. Junqueira, B. C. Reed, and M. Serafini. Zab: High-performance broadcast for primary-backup systems. In *Proceedings of the 2011 IEEE/IFIP 41st International Conference on Dependable Systems&Networks*, DSN '11, Washington, DC, USA, 2011. IEEE Computer Society.

[28] A. Kalia, M. Kaminsky, and D. G. Andersen. Design guidelines for high performance RDMA systems. In *2016 USENIX Annual Technical Conference (USENIX ATC 16)*, Denver, CO, June 2016. USENIX Association.

[29] 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.

[30] E. Kohler, R. Morris, B. Chen, J. Jannotti, and M. F. Kaashoek. The Click modular router. *ACM Transactions on Computer Systems*, 18(3), Aug. 2000.

[31] T. Koponen, M. Casado, N. Gude, J. Stribling, L. Poutievski, M. Zhu, R. Ramanathan, Y. Iwata, H. Inoue, T. Hama, and S. Shenker. Onix: A distributed control platform for large-scale production networks. In *Proceedings of the 9th USENIX Symposium on Operating Systems Design and Implementation (OSDI '10)*, Vancouver, BC, Canada, Oct. 2010. USENIX.

[32] 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.

[33] L. Lamport. The part-time parliament. *ACM Transactions on Computer Systems*, 16(2), May 1998.

[34] L. Lamport. Paxos made simple. *ACM SIGACT News*, 32(4), Dec. 2001.

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

[36] L. Lamport. Fast Paxos. *Distributed Computing*, 19(2), Oct. 2006.

[37] L. Lamport. Lower bounds for asynchronous consensus. *Distributed Computing*, 19(2), Oct. 2006.

[38] C. Leau. Spring Data Redis – Retwis-J, 2013. `http://docs.spring.io/spring-data/data-keyvalue/examples/retwisj/current/`.

[39] J. Li, E. Michael, N. K. Sharma, A. Szekeres, and D. R. K. Ports. Just say NO to Paxos overhead: Replacing consensus with network ordering [extended version]. Technical Report UW-CSE-16-09-02, University of Washington CSE, Seattle, WA, USA, Nov. 2016.

[40] 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 *Proceedings of the 5th Symposium on Cloud Computing (SOCC '14)*, Seattle, WA, USA, Nov. 2014. ACM.

[41] X. Li, R. Sethi, M. Kaminsky, D. G. Andersen, and M. J. Freedman. Be fast, cheap and in control with SwitchKV. In *13th USENIX Symposium on Networked Systems Design and Implementation (NSDI 16)*, Santa Clara, CA, Mar. 2016. USENIX Association.

[42] 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.

[43] 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.

[44] 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), Apr. 2008.

[45] I. Moraru, D. G. Andersen, and M. Kaminsky. There is more consensus in egalitarian parliaments. In *Proc. of SOSP*, 2013.

[46] 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.

[47] J. Naous, D. Erickson, G. A. Covington, G. Appenzeller, and N. McKeown. Implementing an OpenFlow switch on the NetFPGA platform. In *Proceedings of the 4th ACM/IEEE Symposium on Architectures for Networking and Communications Systems*, ANCS '08, New York, NY, USA, 2008. ACM.

[48] 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.

[49] R. Ozdag. Intel® Ethernet switch FM6000 series-software defined networking.

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

[51] F. Pedone and A. Schiper. Optimistic atomic broadcast: A pragmatic viewpoint. *Theor. Comput. Sci.*, 291(1), Jan. 2003.

[52] S. Peter, J. Li, I. Zhang, D. R. K. Ports, D. Woos, A. Krishnamurthy, T. Anderson, and T. Roscoe. Arrakis: The operating system is the control plane. In *Proceedings of the 11th USENIX Symposium on Operating Systems Design and Implementation (OSDI '14)*, Broomfield, CO, USA, Oct. 2014. USENIX.

[53] D. R. K. Ports, J. Li, V. Liu, N. K. Sharma, and A. Krishnamurthy. Designing distributed systems using approximate synchrony in data center networks. In *Proc. of NSDI*, 2015.

[54] J. Rao, E. J. Shekita, and S. Tata. Using Paxos to build a scalable, consistent, and highly available datastore. *Proc. of VLDB*, 4(4), Apr. 2011.

[55] J. H. Saltzer, D. P. Reed, and D. D. Clark. End-to-end arguments in system design. *ACM Transactions on Computer Systems*, 2(4), Nov. 1984.

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

[57] T. Spalink, S. Karlin, L. Peterson, and Y. Gottlieb. Building a robust software-based router using network processors. In *Proceedings of the 19th ACM Symposium on Operating Systems Principles (SOSP '01)*, Banff, Canada, Oct. 2001. ACM.

[58] 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.

[59] XPliant Ethernet switch product family. `www.cavium.com/XPliant-Ethernet-Switch-Product-Family.html`.

[60] I. Zhang, N. K. Sharma, A. Szekeres, A. Krishnamurthy, and D. R. K. Ports. Building consistent transactions with inconsistent replication. In *Proc. of SOSP*, 2015.

XFT: Practical Fault Tolerance Beyond Crashes

Shengyun Liu
NUDT[*]

Paolo Viotti
EURECOM

Christian Cachin
IBM Research - Zurich

Vivien Quéma
Grenoble INP

Marko Vukolić
IBM Research - Zurich

Abstract

Despite years of intensive research, Byzantine fault-tolerant (BFT) systems have not yet been adopted in practice. This is due to additional cost of BFT in terms of resources, protocol complexity and performance, compared with crash fault-tolerance (CFT). This overhead of BFT comes from the assumption of a powerful adversary that can fully *control* not only the Byzantine faulty machines, but *at the same time* also the message delivery schedule across the *entire* network, effectively inducing communication asynchrony and partitioning otherwise correct machines at will. To many practitioners, however, such strong attacks appear irrelevant.

In this paper, we introduce *cross fault tolerance* or *XFT*, a novel approach to building reliable and secure distributed systems and apply it to the classical state-machine replication (SMR) problem. In short, an XFT SMR protocol provides the reliability guarantees of widely used asynchronous CFT SMR protocols such as Paxos and Raft, but also tolerates Byzantine faults in combination with network asynchrony, as long as a majority of replicas are correct and communicate synchronously. This allows the development of XFT systems at the price of CFT (already paid for in practice), yet with strictly stronger resilience than CFT — sometimes even stronger than BFT itself.

As a showcase for XFT, we present XPaxos, the first XFT SMR protocol, and deploy it in a geo-replicated setting. Although it offers much stronger resilience than CFT SMR at no extra resource cost, the performance of XPaxos matches that of the state-of-the-art CFT protocols.

1 Introduction

Tolerance to any kind of service disruption, whether caused by a simple hardware fault or by a large-scale disaster, is key for the survival of modern distributed systems. Cloud-scale applications must be inherently resilient, as any outage has direct implications on the business behind them [24].

Modern production systems (e.g., [13, 8]) increase the number of *nines of reliability*[1] by employing sophisticated distributed protocols that tolerate *crash* machine faults as well as *network faults*, such as network partitions or asynchrony, which reflect the inability of otherwise *correct* machines to communicate among each other in a timely manner. At the heart of these systems typically lies a crash fault-tolerant (CFT) consensus-based *state-machine replication* (SMR) primitive [36, 10].

These systems cannot deal with *non-crash* (or *Byzantine* [29]) faults, which include not only malicious, adversarial behavior, but also arise from errors in the hardware, stale or corrupted data from storage systems, memory errors caused by physical effects, bugs in software, hardware faults due to ever smaller circuits, and human mistakes that cause state corruptions and data loss. However, such problems do occur in practice — each of these faults has a public record of taking down major production systems and corrupting their service [14, 4].

Despite more than 30 years of intensive research since the seminal work of Lamport, Shostak and Pease [29], no *practical* answer to tolerating non-crash faults has emerged so far. In particular, asynchronous Byzantine fault-tolerance (BFT), which promises to resolve this problem [9], has not lived up to this expectation, largely because of its extra cost compared with CFT. Namely, asynchronous (that is, "eventually synchronous" [18]) BFT SMR must use at least $3t+1$ replicas to tolerate t non-crash faults [7] instead of only $2t+1$ replicas for CFT, as used by Paxos [27] or Raft [34], for example.

The overhead of asynchronous BFT is due to the extraordinary power given to the adversary, which may

[*]Work done while being a PhD student at EURECOM.

[1]As an illustration, five nines reliability means that a system is up and correctly running at least 99.999% of the time. In other words, malfunction is limited to one hour every 10 years on average.

control both the Byzantine faulty machines *and* the *entire network* in a coordinated way. In particular, the classical BFT adversary can partition *any number* of otherwise correct machines at will. In line with observations by practitioners [25], we claim that this adversary model is actually too strong for the phenomena observed in deployed systems. For instance, accidental non-crash faults usually do not lead to network partitions. Even malicious non-crash faults rarely cause the whole network to break down in wide-area networks and geo-replicated systems. The proverbial all-powerful attacker as a common source behind those faults is a popular and powerful simplification used for the design phase, but it has not seen equivalent proliferation in practice.

In this paper, we introduce *XFT* (short for *cross fault tolerance*), a novel approach to building efficient resilient distributed systems that tolerate both non-crash (Byzantine) faults and network faults (asynchrony). In short, XFT allows building resilient systems that

- do not use extra resources (replicas) compared with asynchronous CFT;
- preserve *all* reliability guarantees of asynchronous CFT (that is, in the absence of Byzantine faults); and
- provide correct service (i.e., safety and liveness [2]) even when Byzantine faults do occur, as long as a majority of the replicas are correct and can communicate with each other synchronously (that is, when a minority of the replicas are Byzantine-faulty or partitioned because of a network fault).

In particular, we envision XFT for wide-area or *geo-replicated* systems [13], as well as for any other deployment where an adversary cannot easily coordinate enough network partitions and Byzantine-faulty machine actions at the same time.

As a showcase for XFT, we present XPaxos, the first state-machine replication protocol in the XFT model. XPaxos tolerates faults beyond crashes in an efficient and practical way, achieving much greater coverage of realistic failure scenarios than the state-of-the-art CFT SMR protocols, such as Paxos or Raft. This comes without resource overhead as XPaxos uses $2t+1$ replicas. To validate the performance of XPaxos, we deployed it in a geo-replicated setting across Amazon EC2 datacenters worldwide. In particular, we integrated XPaxos within Apache ZooKeeper, a prominent and widely used coordination service for cloud systems [19]. Our evaluation on EC2 shows that XPaxos performs almost as well in terms of throughput and latency as a WAN-optimized variant of Paxos, and significantly better than the best available BFT protocols. In our evaluation, XPaxos even outperforms the native CFT SMR protocol built into ZooKeeper [20].

Finally, and perhaps surprisingly, we show that XFT can offer *strictly stronger* reliability guarantees than state-of-the-art BFT, for instance under the assumption that machine faults and network faults occur as independent and identically distributed random variables, for certain probabilities. To this end, we calculate the number of nines of consistency (system safety) and availability (system liveness) of resource-optimal CFT, BFT and XFT (e.g., XPaxos) protocols. Whereas XFT *always* provides strictly stronger consistency and availability guarantees than CFT and *always* strictly stronger availability guarantees than BFT, our reliability analysis shows that, in some cases, XFT also provides strictly stronger consistency guarantees than BFT.

The remainder of this paper is organized as follows. In Section 2, we define the system model, which is then followed by the definition of the XFT model in Section 3. In Section 4 and Section 5, we present XPaxos and its evaluation in the geo-replicated context, respectively. Section 6 provides simplified reliability analysis comparing XFT with CFT and BFT. We overview related work and conclude in Section 7. For space reasons, the full correctness proof of XPaxos is given in [31].

2 System model

Machines. We consider a message-passing distributed system containing a set Π of $n = |\Pi|$ *machines*, also called *replicas*. Additionally, there is a separate set C of *client* machines.

Clients and replicas may suffer from Byzantine *faults*: we distinguish between *crash* faults, where a machine simply stops all computation and communication, and *non-crash* faults, where a machine acts arbitrarily, but cannot break cryptographic primitives we use (cryptographic hashes, MACs, message digests and digital signatures). A machine that is not faulty is called *correct*. We say a machine is *benign* if the machine is correct or crash-faulty. We further denote the number of replica faults at a given moment s by

- $t_c(s)$: the number of crash-faulty replicas, and
- $t_{nc}(s)$: the number of non-crash-faulty replicas.

Network. Each pair of replicas is connected with reliable point-to-point bi-directional communication channels. In addition, each client can communicate with any replica.

The system can be *asynchronous* in the sense that machines may not be able to exchange messages and obtain responses to their requests in time. In other words, *network faults* are possible; we define a *network fault* as the inability of some *correct* replicas to communicate with each other in a timely manner, that is, when a message exchanged between two correct replicas cannot be delivered and processed within delay Δ, known to all replicas. Note that Δ is a deployment specific parameter: we discuss practical choices for Δ in the context of our geo-replicated setting in Section 5. Finally, we assume an *eventually synchronous* system in which, eventually, network faults do not occur [18].

Note that we model an excessive processing delay as a network problem and *not* as an issue related to a machine fault. This choice is made consciously, rooted in the experience that for the general class of protocols considered in this work, a long local processing time is never an issue on correct machines compared with network delays.

To help quantify the number of network faults, we first give the definition of partitioned replica.

Definition 1 (Partitioned replica)**.** *Replica p is partitioned if p is* ***not*** *in the largest subset of replicas, in which every pair of replicas can communicate among each other within delay* Δ.

If there is more than one subset with the maximum size, only one of them is recognized as the largest subset. For example in Figure 1, the number of partitioned replicas is 3, counting either the group of p_1, p_4 and p_5 or that of p_2, p_3 and p_5. The number of partitioned replicas can be as much as $n-1$, which means that no two replicas can communicate with each other within delay Δ. We say replica p is *synchronous* if p is not partitioned. We now quantify network faults at a given moment s as

- $t_p(s)$: the number of correct, but partitioned replicas.

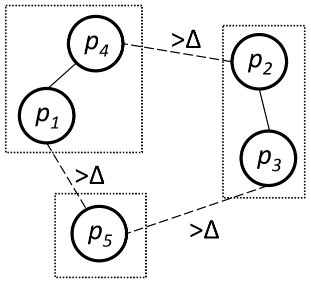


Figure 1: An illustration of partitioned replicas: $\{p_1, p_4, p_5\}$ or $\{p_2, p_3, p_5\}$ are partitioned based on Definition 1.

Problem. In this paper, we focus on the *deterministic* state-machine replication problem (SMR) [36]. In short, in SMR clients invoke requests, which are then committed by replicas. SMR ensures

- *safety*, or *consistency*, by (a) enforcing *total order* across committed client's *requests* across all correct replicas; and by (b) enforcing *validity*, i.e., that a correct replica commits a request only if it was previously invoked by a client;
- *liveness*, or *availability*, by eventually committing a request by a correct client at all correct replicas and returning an application-level reply to the client.

3 The XFT model

This section introduces the XFT model and relates it to the established crash-fault tolerance (CFT) and Byzantine-fault tolerance (BFT) models.

3.1 XFT in a nutshell

Classical CFT and BFT explicitly model machine faults only. These are then combined with an orthogonal network fault model, either the synchronous model (where network faults in our sense are ruled out), or the asynchronous model (which includes *any number* of network faults). Hence, previous work can be classified into four categories: synchronous CFT [16, 36], asynchronous CFT [36, 27, 33], synchronous BFT [29, 17, 6], and asynchronous BFT [9, 3].

XFT, in contrast, redefines the boundaries between machine and network fault dimensions: XFT allows the design of reliable protocols that tolerate crash machine faults regardless of the number of network faults and that, at the same time, tolerate non-crash machine faults when the number of machines that are either faulty or partitioned is within a threshold.

To formalize XFT, we first define *anarchy*, a very severe system condition with actual non-crash machine (replica) faults and plenty of faults of different kinds, as follows:

Definition 2 (Anarchy)**.** *The system is in anarchy at a given moment s iff* $t_{nc}(s) > 0$ *and* $t_c(s) + t_{nc}(s) + t_p(s) > t$.

Here, t is the threshold of replica faults, such that $t \leq \lfloor \frac{n-1}{2} \rfloor$. In other words, in anarchy, some replica is non-crash-faulty, and there is no correct and synchronous majority of replicas. Armed with the definition of anarchy, we can define XFT protocols for an arbitrary distributed computing problem in function of its safety property [2].

Definition 3 (XFT protocol)**.** *Protocol P is an XFT protocol if P satisfies safety in all executions in which the system is never in anarchy.*

Liveness of an XFT protocol will typically depend on a problem and implementation. For instance, for deterministic SMR we consider in this paper, our XPaxos

		Maximum number of each type of replica faults		
		non-crash faults	crash faults	partitioned replicas
Asynchronous CFT (e.g., Paxos [28])	consistency	0	n	$n-1$
	availability	0	$\lfloor \frac{n-1}{2} \rfloor$ (combined)	
Asynchronous BFT (e.g., PBFT [9])	consistency	$\lfloor \frac{n-1}{3} \rfloor$	n	$n-1$
	availability	$\lfloor \frac{n-1}{3} \rfloor$ (combined)		
(Authenticated) Synchronous BFT (e.g., [29])	consistency	$n-1$	n	0
	availability	$n-1$ (combined)		0
XFT (e.g., XPaxos)	consistency	0	n	$n-1$
		$\lfloor \frac{n-1}{2} \rfloor$ (combined)		
	availability	$\lfloor \frac{n-1}{2} \rfloor$ (combined)		

Table 1: The maximum numbers of each type of fault tolerated by representative SMR protocols. Note that XFT provides consistency in two modes, depending on the occurrence of non-crash faults.

protocol eventually satisfies liveness, provided a majority of replicas is correct and synchronous. This can be shown optimal.

3.2 XFT vs. CFT/BFT

Table 1 illustrates differences between XFT and CFT/BFT in terms of their consistency and availability guarantees for SMR.

State-of-the-art asynchronous CFT protocols [28, 34] guarantee consistency despite *any* number of crash-faulty replicas and *any* number of partitioned replicas. They also guarantee availability whenever a majority of replicas ($t \leq \lfloor \frac{n-1}{2} \rfloor$) are correct and synchronous. As soon as a single machine is non-crash-faulty, CFT protocols guarantee neither consistency nor availability.

Optimal asynchronous BFT protocols [9, 22, 3] guarantee consistency despite any number of crash-faulty or partitioned replicas, with at most $t = \lfloor \frac{n-1}{3} \rfloor$ non-crash-faulty replicas. They also guarantee availability with up to $\lfloor \frac{n-1}{3} \rfloor$ combined faults, i.e., whenever more than two-thirds of replicas are correct and not partitioned. Note that BFT availability might be weaker than that of CFT in the absence of non-crash faults — unlike CFT, BFT does not guarantee availability when the sum of crash-faulty and partitioned replicas is in the range $[n/3, n/2)$.

Synchronous BFT protocols (e.g., [29]) do not consider the existence of correct, but partitioned replicas. This makes for a very strong assumption — and helps synchronous BFT protocols that use digital signatures for message authentication (so called *authenticated* protocols) to tolerate up to $n-1$ non-crash-faulty replicas.

In contrast, XFT protocols with optimal resilience, such as our XPaxos, guarantee consistency in two modes: *(i)* without non-crash faults, despite any number of crash-faulty and partitioned replicas (i.e., just like CFT), and *(ii)* with non-crash faults, whenever a majority of replicas are correct and not partitioned, i.e., provided the sum of all kinds of faults (machine or network faults) does not exceed $\lfloor \frac{n-1}{2} \rfloor$. Similarly, it also guarantees availability whenever a majority of replicas are correct and not partitioned.

It may be tempting to view XFT as some sort of a combination of the asynchronous CFT and synchronous BFT models. However, this is misleading, as even with actual non-crash faults, XFT is incomparable to authenticated synchronous BFT. Specifically, authenticated synchronous BFT protocols, such as the seminal Byzantine Generals protocol [29], may violate consistency with a single partitioned replica. For instance, with $n = 5$ replicas and an execution in which three replicas are correct and synchronous, one replica is correct but partitioned and one replica is non-crash-faulty, the XFT model mandates that the consistency be preserved, whereas the Byzantine Generals protocol may violate consistency.[2]

Furthermore, from Table 1, it is evident that XFT offers strictly stronger guarantees than asynchronous CFT, for both availability and consistency. XFT also offers strictly stronger availability guarantees than asynchronous BFT. Finally, the consistency guarantees of XFT are incomparable to those of asynchronous BFT. On the one hand, outside anarchy, XFT is consistent with the number of non-crash faults in the range $[n/3, n/2)$, whereas asynchronous BFT is not. On the other hand, unlike XFT, asynchronous BFT is consistent in anarchy

[2]XFT is not stronger than authenticated synchronous BFT either, as the latter tolerates more machine faults in the complete absence of network faults.

provided the number of non-crash faults is less than $n/3$. We discuss these points further in Section 6, where we also quantify the reliability comparison between XFT and asynchronous CFT/BFT assuming the special case of independent faults.

3.3 Where to use XFT?

The intuition behind XFT starts from the assumption that "extremely bad" system conditions, such as anarchy, are very rare, and that providing consistency guarantees in anarchy might not be worth paying the asynchronous BFT premium.

In practice, this assumption is plausible in many deployments. We envision XFT for use cases in which an adversary cannot easily coordinate enough network partitions and non-crash-faulty machine actions at the same time. Some interesting candidate use cases include:

- *Tolerating "accidental" non-crash faults.* In systems which are not susceptible to malicious behavior and deliberate attacks, XFT can be used to protect against "accidental" non-crash faults, which can be assumed to be largely independent of network faults. In such cases, XFT could be used to harden CFT systems without considerable overhead of BFT.

- *Wide-area networks and geo-replicated systems.* XFT may reveal useful even in cases where the system is susceptible to malicious non-crash faults, as long as it may be difficult or expensive for an adversary to coordinate an attack to compromise Byzantine machines and partition sufficiently many replicas *at the same time*. Particularly interesting for XFT are WAN and geo-replicated systems which often enjoy redundant communication paths and typically have a smaller surface for network-level DoS attacks (e.g., no multicast storms and flooding).

- *Blockchain.* A special case of geo-replicated systems, interesting to XFT, are blockchain systems. In a typical blockchain system, such as Bitcoin [32], participants may be financially motivated to act maliciously, yet may lack the means and capabilities to compromise the communication among (a large number of) correct participants. In this context, XFT is particularly interesting for so-called permissioned blockchains, which are based on state-machine replication rather than on Bitcoin-style proof-of-work [40].

4 XPaxos Protocol

4.1 XPaxos overview

XPaxos is a novel state-machine replication (SMR) protocol designed specifically in the XFT model. XPaxos specifically targets good performance in geo-replicated settings, which are characterized by the network being the bottleneck, with high link latency and relatively low, heterogeneous link bandwidth.

In a nutshell, XPaxos consists of three main components:

- A common-case protocol, which replicates and totally orders requests across replicas. This has, roughly speaking, the message pattern and complexity of communication among replicas of state-of-the-art CFT protocols (e.g., Phase 2 of Paxos), hardened by the use of digital signatures.

- A novel view-change protocol, in which the information is transferred from one view (system configuration) to another in a *decentralized*, leaderless fashion.

- A fault detection (FD) mechanism, which can help detect, outside anarchy, non-crash faults that would leave the system in an inconsistent state in anarchy. The goal of the FD mechanism is to minimize the impact of long-lived non-crash faults (in particular "data loss" faults) in the system and to help detect them before they coincide with a sufficient number of crash faults and network faults to push the system into anarchy.

XPaxos is orchestrated in a sequence of *views* [9]. The central idea in XPaxos is that, during common-case operation in a given view, XPaxos synchronously replicates clients' requests to only $t+1$ replicas, which are the members of a *synchronous group* (out of $n = 2t+1$ replicas in total). Each view number i uniquely determines the synchronous group, sg_i, using a mapping known to all replicas. Every synchronous group consists of one *primary* and t *followers*, which are jointly called *active replicas*. The remaining t replicas in a given view are called *passive* replicas; optionally, passive replicas learn the order from the active replicas using the *lazy replication* approach [26]. A view is not changed unless there is a machine or network fault within the synchronous group.

In the common case (Section 4.2), the clients send digitally signed requests to the primary, which are then replicated across $t+1$ active replicas. These $t+1$ replicas digitally sign and locally log the proofs for all replicated requests to their *commit logs*. Commit logs then serve as the basis for maintaining consistency in view changes.

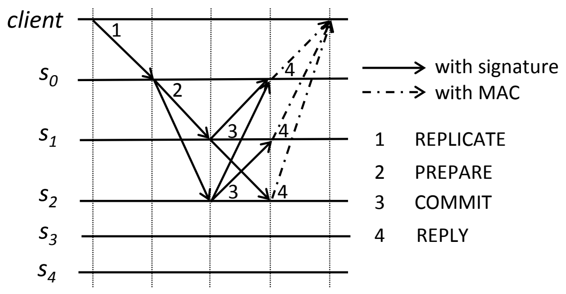
The view change of XPaxos (Section 4.3) reconfigures the entire synchronous group, not only the leader. All $t+1$ active replicas of the new synchronous group sg_{i+1} try to transfer the state from the preceding views to view $i+1$. This *decentralized* approach to view change stands in sharp contrast to the classical reconfiguration/view-change in CFT and BFT protocols (e.g., [27, 9]), in which only a single replica (the primary) leads the view change and transfers the state from previous views. This difference is crucial to maintaining consistency (i.e., total order) across XPaxos views in the presence of non-crash faults (but in the absence of full anarchy). This novel and decentralized view-change scheme of XPaxos guarantees that even in the presence of non-crash faults, but outside anarchy, at least one correct replica from the new synchronous group sg_{i+1} will be able to transfer the correct state from previous views, as it will be able to contact some correct replica from any old synchronous group.

Finally, the main idea behind the FD scheme of XPaxos is the following. In view change, a non-crash-faulty replica (of an old synchronous group) might not transfer its latest state to a correct replica in the new synchronous group. This "data loss" fault is dangerous, as it may violate consistency when the system is in anarchy. However, such a fault can be detected using digital signatures from the commit log of some correct replicas (from an old synchronous group), provided that these correct replicas can communicate synchronously with correct replicas from the new synchronous group. In a sense, with XPaxos FD, a critical non-crash machine fault must occur for the first time *together* with sufficiently many crash or partitioned machines (i.e., in anarchy) to violate consistency.

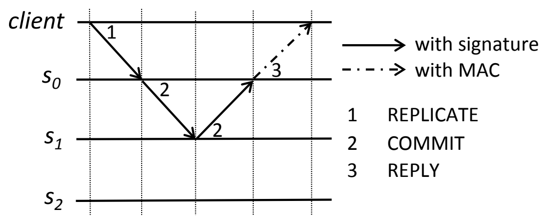
In the following, we explain the core of XPaxos for the common case (Section 4.2), view-change (Section 4.3) and fault detection (Section 4.4) components. We discuss XPaxos optimizations in Section 4.5 and give XPaxos correctness arguments in Section 4.6. Because of space limitations, the complete pseudocode and correctness proof have been included in [31].

4.2 Common case

Figure 2 shows the common-case message patterns of XPaxos for the general case ($t \geq 2$) and for the special case $t = 1$. XPaxos is specifically optimized for the case where $t = 1$, as in this case, there are only two active replicas in each view and the protocol is very efficient. The special case $t = 1$ is also highly relevant in practice (see e.g., Spanner [13]). In the following, we first explain XPaxos in the general case, and then focus on the $t = 1$ special case.



(a) $t \geq 2$



(b) $t = 1$

Figure 2: XPaxos common-case message patterns (a) for the general case when $t \geq 2$ and (b) for the special case of $t = 1$. The synchronous groups are (s_0, s_1, s_2) and (s_0, s_1), respectively.

Notation. We denote the digest of a message m by $D(m)$, whereas $\langle m \rangle_{\sigma_p}$ denotes a message that contains both $D(m)$ signed by the private key of machine p and m. For signature verification, we assume that all machines have public keys of all other processes.

4.2.1 General case ($t \geq 2$)

The common-case message pattern of XPaxos is shown in Figure 2a. More specifically, upon receiving a signed request $req = \langle \text{REPLICATE}, op, ts_c, c \rangle_{\sigma_c}$ from client c (where op is the client's operation and ts_c is the client's timestamp), the primary (say s_0) (1) increments sequence number sn and assigns sn to req, (2) signs a message $prep = \langle \text{PREPARE}, D(req), sn, i \rangle_{\sigma_{s_0}}$ and logs $\langle req, prep \rangle$ into its prepare log $PrepareLog_0[sn]$ (we say s_0 *prepares* req), and (3) forwards $\langle req, prep \rangle$ to *all other active replicas* (i.e, the t followers).

Each follower s_j ($1 \leq j \leq t$) verifies the primary's and client's signatures, checks whether its local sequence number equals $sn - 1$, and logs $\langle req, prep \rangle$ into its prepare log $PrepareLog_j[sn]$. Then, s_j updates its local sequence number to sn, *signs* the digest of the request req, the sequence number sn and the view number i, and sends $\langle \text{COMMIT}, D(req), sn, i \rangle_{\sigma_{s_j}}$ to all active replicas.

Upon receiving t signed COMMIT messages — one from each follower — such that a matching entry is in the prepare log, an active replica s_k ($0 \leq k \leq t$) logs *prep* and the t signed COMMIT messages into its commit log $CommitLog_{s_k}[sn]$. We say s_k *commits req* when this occurs. Finally, s_k executes *req* and sends the authenticated reply to the client (followers may only send the digest of the reply). The client commits the request when it receives matching REPLY messages from all $t+1$ active replicas.

A client that times out without committing the requests broadcasts the request to all active replicas. Active replicas then forward such a request to the primary and trigger a *retransmission timer*, within which a correct active replica expects the client's request to be committed.

4.2.2 Tolerating a single fault ($t = 1$).

When $t = 1$, the XPaxos common case simplifies to involving only 2 messages between 2 active replicas (see Figure 2b).

Upon receiving a signed request $req = \langle \text{REPLICATE}, op, ts_c, c \rangle_{\sigma_c}$ from client c, the primary (s_0) increments the sequence number sn, signs sn along the digest of *req* and view number i in message $m_0 = \langle \text{COMMIT}, D(req), sn, i \rangle_{\sigma_{s_0}}$, stores $\langle req, m_0 \rangle$ into its prepare log ($PrepareLog_{s_0}[sn] = \langle req, m_0 \rangle$), and sends the message $\langle req, m_0 \rangle$ to the follower s_1.

On receiving $\langle req, m_0 \rangle$, the follower s_1 verifies the client's and primary's signatures, and checks whether its local sequence number equals $sn - 1$. If so, the follower updates its local sequence number to sn, executes the request producing reply $R(req)$, and signs message m_1; m_1 is similar to m_0, but also includes the client's timestamp and the digest of the reply: $m_1 = \langle \text{COMMIT}, \langle D(req), sn, i, req.ts_c, D(R(req)) \rangle_{\sigma_{s_1}}$. The follower then saves the tuple $\langle req, m_0, m_1 \rangle$ to its commit log ($CommitLog_{s_1}[sn] = \langle req, m_0, m_1 \rangle$) and sends m_1 to the primary.

The primary, on receiving a valid COMMIT message from the follower (with a matching entry in its prepare log), executes the request, compares the reply $R(req)$ with the follower's digest contained in m_1, and stores $\langle req, m_0, m_1 \rangle$ in its commit log. Finally, it returns an authenticated reply containing m_1 to c, which commits the request if all digests and the follower's signature match.

4.3 View change

Intuition. The ordered requests in commit logs of correct replicas are the key to enforcing consistency (total order) in XPaxos. To illustrate an XPaxos view change, consider synchronous groups sg_i and sg_{i+1} of views i and $i+1$, respectively, each containing $t+1$ replicas. Note that proofs of requests committed in sg_i might have been logged by *only one* correct replica in sg_i. Nevertheless, the XPaxos view change must ensure that (outside anarchy) these proofs are transferred to the new view $i+1$. To this end, we had to depart from traditional view change

		Synchronous Groups ($i \in \mathbb{N}_0$)		
		sg_i	sg_{i+1}	sg_{i+2}
Active replicas	Primary	s_0	s_0	s_1
	Follower	s_1	s_2	s_2
Passive replica		s_2	s_1	s_0

Table 2: Synchronous group combinations ($t = 1$).

techniques [9, 22, 12] where the entire view-change is led by a single replica, usually the primary of the new view. Instead, in XPaxos view change, *every active replica in* sg_{i+1} *retrieves information about requests committed in preceding views.* Intuitively, with correct majority of correct and synchronous replicas, at least one correct and synchronous replica from sg_{i+1} will contact (at least one) correct and synchronous replica from sg_i and transfer the latest correct commit log to the new view $i+1$.

In the following, we first describe how we choose active replicas for each view. Then, we explain how view changes are initiated, and, finally, how view changes are performed.

4.3.1 Choosing active replicas

To choose active replicas for view i, we may enumerate all sets containing $t+1$ replicas (i.e., $\binom{2t+1}{t+1}$ sets) which then alternate as synchronous groups across views in a round-robin fashion. In addition, each synchronous group uniquely determines the primary. We assume that the mapping from view numbers to synchronous groups is known to all replicas (see e.g., Table 2).

The above simple scheme works well for small number of replicas (e.g., $t = 1$ and $t = 2$). For a large number of replicas, the combinatorial number of synchronous groups may be inefficient. To this end, XPaxos can be modified to rotate only the leader, which may then resort to deterministic verifiable pseudorandom selection of the set of f followers in each view. The exact details of such a scheme would, however, exceed the scope of this paper.

4.3.2 View-change initiation

If a synchronous group in view i (denoted by sg_i) does not make progress, XPaxos performs a view change. Only an active replica of sg_i may initiate a view change.

An active replica $s_j \in sg_i$ initiates a view change if (i) s_j receives a message from another active replica that does not conform to the protocol (e.g., an invalid signature), (ii) the retransmission timer at s_j expires, (iii) s_j does not complete a view change to view i in a timely manner, or (iv) s_j receives a valid SUSPECT message for view i from another replica in sg_i. Upon a view-change initiation, s_j stops participating in the current view and sends $\langle \text{SUSPECT}, i, s_j \rangle_{\sigma_{s_j}}$ to *all* other replicas.

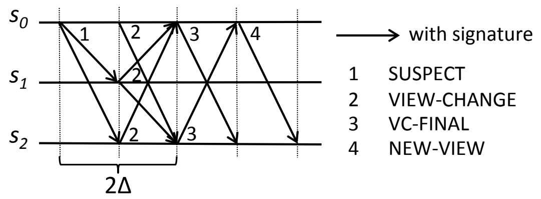


Figure 3: Illustration of XPaxos view change: the synchronous group is changed from (s_0,s_1) to (s_0,s_2).

4.3.3 Performing the view change

Upon receiving a SUSPECT message from an active replica in view i (see the message pattern in Figure 3), replica s_j stops processing messages of view i and sends $m = \langle \text{VIEW-CHANGE}, i+1, s_j, CommitLog_{s_j} \rangle_{\sigma_{s_j}}$ to the $t+1$ active replicas of sg_{i+1}. A VIEW-CHANGE message contains the commit log $CommitLog_{s_j}$ of s_j. Commit logs might be empty (e.g., if s_j was passive).

Note that XPaxos requires all active replicas in the new view to collect the most recent state and its proof (i.e., VIEW-CHANGE messages), rather than only the new primary. Otherwise, a faulty new primary could, even outside anarchy, purposely omit VIEW-CHANGE messages that contain the most recent state. Active replica s_j in view $i+1$ waits for at least $n-t$ VIEW-CHANGE messages from all, but also waits for 2Δ time, trying to collect as many messages as possible.

Upon completion of the above protocol, each active replica $s_j \in sg_{i+1}$ inserts all VIEW-CHANGE messages it has received into set $VCSet^{i+1}_{s_j}$. Then s_j sends $\langle \text{VC-FINAL}, i+1, s_j, VCSet^{i+1}_{s_j} \rangle_{\sigma_{s_j}}$ to every active replica in view $i+1$. This serves to exchange the received VIEW-CHANGE messages among active replicas.

Every active replica $s_j \in sg_{i+1}$ must receive VC-FINAL messages from *all* active replicas in sg_{i+1}, after which s_j extends the value $VCSet^{i+1}_{s_j}$ by combining $VCSet^{i+1}_{*}$ sets piggybacked in VC-FINAL messages. Then, for each sequence number sn, an active replica selects the commit log with the highest view number in all VIEW-CHANGE messages, to confirm the committed request at sn.

Afterwards, to prepare and commit the selected requests in view $i+1$, the new primary ps_{i+1} sends $\langle \text{NEW-VIEW}, i+1, PrepareLog \rangle_{\sigma_{ps_{i+1}}}$ to every active replica in sg_{i+1}, where the array $PrepareLog$ contains the prepare logs generated in view $i+1$ for each selected request. Upon receiving a NEW-VIEW message, every active replica $s_j \in sg_{i+1}$ processes the prepare logs in $PrepareLog$ as described in the common case (see Section 4.2).

Finally, every active replica $s_j \in sg_{i+1}$ makes sure that all selected requests in $PrepareLog$ are committed in view $i+1$. When this condition is satisfied, XPaxos can start processing new requests.

4.4 Fault detection

XPaxos does not guarantee consistency in anarchy. Hence, non-crash faults could violate XPaxos consistency in the long run, if they persist long enough to eventually coincide with enough crash or network faults. To cope with long-lived faults, we propose (an otherwise optional) *Fault Detection (FD)* mechanism for XPaxos.

Roughly speaking, FD guarantees the following property: *if a machine p suffers a non-crash fault outside anarchy in a way that would cause inconsistency in anarchy, then XPaxos FD detects p as faulty (outside anarchy).* In other words, any potentially fatal fault that occurs outside anarchy would be detected by XPaxos FD.

Here, we sketch how FD works in the case $t = 1$ (see [31] for details), focusing on detecting a specific non-crash fault that may render XPaxos inconsistent in anarchy — a *data loss* fault by which a non-crash-faulty replica *loses some of its commit log* prior to a view change. Intuitively, data loss faults are dangerous as they cannot be prevented by the straightforward use of digital signatures.

Our FD mechanism entails modifying the XPaxos view change as follows: in addition to exchanging their commit logs, replicas also exchange their prepare logs. Notice that in the case $t = 1$ only the primary maintains a prepare log (see Section 4.2). In the new view, the primary prepares and the follower commits all requests contained in transferred commit and prepare logs.

With the above modification, to violate consistency, a faulty primary (of preceding view i) would need to exhibit a data loss fault in both its commit log and its prepare log. However, such a data loss fault in the primary's prepare log would be detected, outside anarchy, because (i) the (correct) follower of view i would reply in the view change and (ii) an entry in the primary's prepare log causally precedes the respective entry in the follower's commit log. By simply verifying the signatures in the follower's commit log, the fault of a primary is detected. Conversely, a data loss fault in the commit log of the follower of view i is detected outside anarchy by verifying the signatures in the commit log of the primary of view i.

4.5 XPaxos optimizations

Although the common-case and view-change protocols described above are sufficient to guarantee correctness, we applied several standard performance optimizations to XPaxos. These include checkpointing and lazy replication [26] to passive replicas (to help shorten the state transfer during view change) as well as batching and pipelining (to improve the throughput). Below, we provide a brief overview of these optimizations; the details can be found in [31].

Checkpointing. Similarly to other replication protocols, XPaxos includes a checkpointing mechanism that speeds up view changes and allows garbage collection (by shortening commit logs). To that end, every *CHK* requests (where *CHK* is a configurable parameter) XPaxos checkpoints the state within the synchronous group. Then the proof of checkpoint is lazily propagated to passive replicas.

Lazy replication. To speed up the state transfer in view change, every follower in the synchronous group lazily propagates the commit log to one passive replica. With lazy replication, a new active replica, which might be the passive replica in the preceding view, may only need to retrieve the missing state from others during a view change.

Batching and pipelining. To improve the throughput of cryptographic operations, the primary batches several requests when preparing. The primary waits for B requests, then signs the batched request and sends it to every follower. If primary receives less than B requests within a time limit, the primary batches all requests it has received.

4.6 Correctness arguments

Consistency (Total Order). XPaxos enforces the following invariant, which is key to total order.

Lemma 1. *Outside anarchy, if a benign client c commits a request req with sequence number sn in view i, and a benign replica s_k commits the request req' with sn in view $i' > i$, then $req = req'$.*

A benign client c commits request *req* with sequence number *sn* in view i only after c has received matching replies from $t+1$ active replicas in sg_i. This implies that every benign replica in sg_i stores *req* into its commit log under sequence number *sn*. In the following, we focus on the special case where: $i' = i+1$. This serves as the base step for the proof of Lemma 1 by induction across views which we give in [31].

Recall that, in view $i' = i+1$, all (benign) replicas from sg_{i+1} wait for $n-t = t+1$ VIEW-CHANGE messages containing commit logs transferred from other replicas, as well as for the timer set to 2Δ to expire. Then, replicas in sg_{i+1} exchange this information within VC-FINAL messages. Note that, outside anarchy, there exists at least one *correct* and *synchronous* replica in sg_{i+1}, say s_j. Hence, a benign replica s_k that commits req' in view $i+1$ under sequence number *sn* must have had received VC-FINAL from s_j. In turn, s_j waited for $t+1$ VIEW-CHANGE messages (and timer 2Δ), so it received a VIEW-CHANGE message from some correct and synchronous replica $s_x \in sg_i$ (such a replica exists in sg_i as at most t replicas in sg_i are non-crash-faulty or partitioned). As s_x stored *req* under *sn* in its commit log in view i, it forwards this information to s_j in a VIEW-CHANGE message, and s_j forwards this information to s_k within a VC-FINAL. Hence $req = req'$ follows.

Availability. XPaxos availability is guaranteed if the synchronous group contains only correct and synchronous replicas. With eventual synchrony, we can assume that, eventually, there will be no network faults. In addition, with all combinations of $t+1$ replicas rotating in the role of active replicas, XPaxos guarantees that, eventually, view change in XPaxos will complete with $t+1$ *correct* and *synchronous* active replicas.

5 Performance Evaluation

In this section, we evaluate the performance of XPaxos and compare it to that of Zyzzyva [22], PBFT [9] and a WAN-optimized version of Paxos [27], using the Amazon EC2 worldwide cloud platform. We chose geo-replicated, WAN settings as we believe that these are a better fit for protocols that tolerate Byzantine faults, including XFT and BFT. Indeed, in WAN settings *(i)* there is no single point of failure such as a switch interconnecting machines, *(ii)* there are no correlated failures due to, e.g., a power-outage, a storm, or other natural disasters, and *(iii)* it is difficult for the adversary to flood the network, correlating network and non-crash faults (the last point is relevant for XFT).

In the remainder of this section, we first present the experimental setup (Section 5.1), and then evaluate the performance (throughput and latency) in the fault-free scenario (Section 5.2) as well as under faults (Section 5.3). Finally, we perform a performance comparison using a real application, the ZooKeeper coordination service [19] (Section 5.4), by comparing native ZooKeeper to ZooKeeper variants that use the four replication protocols mentioned above.

5.1 Experimental setup

5.1.1 Synchrony and XPaxos

In a practical deployment of XPaxos, a critical parameter is the value of timeout Δ, i.e., the upper bound on the communication delay between any two *correct* machines. If the round-trip time (RTT) between two correct machines takes more than 2Δ, we declare a network fault (see Section 2). Notably, Δ is vital to the XPaxos view-change (Section 4.3).

To understand the value of Δ in our geo-replicated context, we ran a 3-month experiment during which we continuously measured the round-trip latency across six Amazon EC2 datacenters worldwide using TCP ping

	US West (CA)	Europe (EU)	Tokyo (JP)	Sydney (AU)	Sao Paolo (BR)
US East (VA)	88 /1097 /82190 /166390	92 /1112 /85649 /169749	179 /1226 /81177 /165277	268 /1372 /95074 /179174	146 /1214 /85434 /169534
US West (CA)		174 /1184 /1974 /15467	120 /1133 /1180 /6210	186 /1209 /6354 /51646	207 /1252 /90980 /169080
Europe (EU)			287 /1310 /1397 /4798	342 /1375 /3154 /11052	233 /1257 /1382 /9188
Tokyo (JP)				137 /1149 /1414 /5228	394 /2496 /11399 /94775
Sydney (AU)					392 /1496 /2134 /10983

Table 3: Round-trip latency of TCP ping (*hping3*) across Amazon EC2 datacenters, collected during three months. The latencies are given in milliseconds, in the format: average / 99.99% / 99.999% / maximum.

(hping3). We used the least expensive EC2 micro instances, which arguably have the highest probability of experiencing variable latency due to virtualization. Each instance was pinging all other instances every 100 ms. The results of this experiment are summarized in Table 3. While we detected network faults lasting up to 3 min, our experiment showed that the round-trip latency between *any* two datacenters was less than 2.5 sec 99.99% of the time. Therefore, we adopted the value of $\Delta = 2.5/2 = 1.25$ sec.

5.1.2 Protocols under test

We compare XPaxos with three protocols whose common-case message patterns when $t = 1$ are shown in Figure 4. The first two are BFT protocols, namely (a speculative variant of) PBFT [9] and Zyzzyva [22], and require $3t+1$ replicas to tolerate t faults. We chose PBFT because it is possible to derive a speculative variant of the protocol that relies on a 2-phase common-case commit protocol across only $2t+1$ replicas (Figure 4a; see also [9]). In this PBFT variant, the remaining t replicas are not involved in the common case, which is more efficient in a geo-replicated settings. We chose Zyzzyva because it is the fastest BFT protocol that involves all replicas in the common case (Figure 4b). The third protocol we compare against is a very efficient WAN-optimized variant of crash-tolerant Paxos inspired by [5, 23, 13]. We have chosen the variant of Paxos that exhibits the fastest write pattern (Figure 4c). This variant requires $2t+1$ replicas to tolerate t faults, but involves $t+1$ replicas in the common case, i.e., just like XPaxos.

To provide a fair comparison, all protocols rely on the same Java code base and use batching, with the batch size set to 20. We rely on HMAC-SHA1 to compute MACs and RSA1024 to sign and verify signatures computed using the Crypto++ [1] library that we interface with the various protocols using JNI.

5.1.3 Experimental testbed and benchmarks

We run the experiments on the Amazon EC2 platform which comprises widely distributed datacenters, interconnected by the Internet. Communications between datacenters have a low bandwidth and a high latency. We run the experiments on mid-range virtual machines that

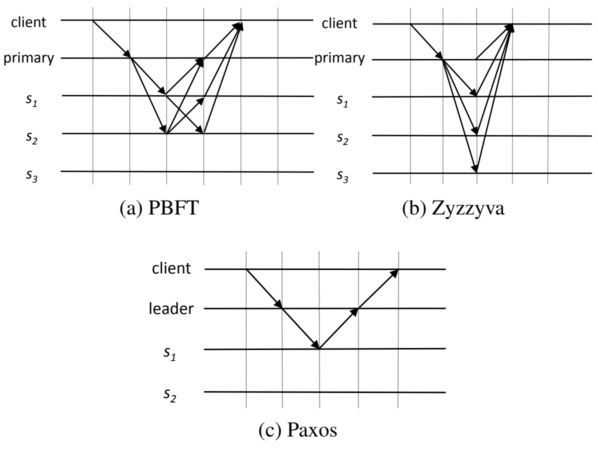


Figure 4: Communication patterns of the three protocols under test ($t = 1$).

contain 8 vCPUs, 15 GB of memory, 2 x 80 GB SSD storage, and run Ubuntu Server 14.04 LTS (PV) with the Linux 3.13.0-24-generic x86_64 kernel.

In the case $t = 1$, Table 4 gives the deployment of the different replicas at different datacenters, for each protocol analyzed. Clients are always located in the same datacenter as the (initial) primary to better emulate what is done in modern geo-replicated systems where clients are served by the closest datacenter [37, 13].[3]

To stress the protocols, we run a microbenchmark that is similar to the one used in [9, 22]. In this microbenchmark, each server replicates a *null* service (this means that there is no execution of requests). Moreover, clients issue requests in *closed-loop*: a client waits for a reply to its current request before issuing a new request. The benchmark allows both the request size and the reply size to be varied. For space limitations, we only report results for two request sizes (1kB, 4kB) and one reply size (0kB). We refer to these microbenchmarks as 1/0 and 4/0 benchmarks, respectively.

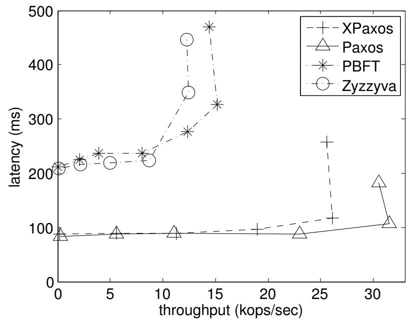
[3]In practice, modern geo-replicated system, like Spanner [13], use hundreds of CFT SMR instances across different partitions to accommodate geo-distributed clients.

5.2 Fault-free performance

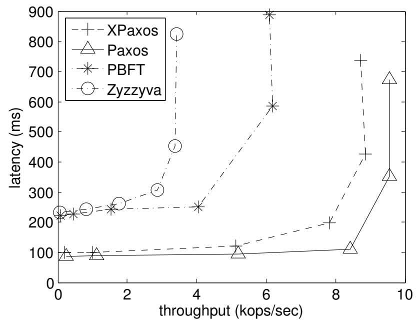
We first compare the performance of protocols when $t = 1$ in replica configurations as shown in Table 4, using the 1/0 and 4/0 microbenchmarks. The results are shown in Figures 5a and 5b. In each graph, the X-axis shows the throughput (in kops/sec), and Y-axis the latency (in ms).

PBFT	Zyzzyva	Paxos	XPaxos	EC2 Region
Primary	Primary	Primary	Primary	US West (CA)
Active	Active	Active	Follower	US East (VA)
		Passive	Passive	Tokyo (JP)
Passive		-	-	Europe (EU)

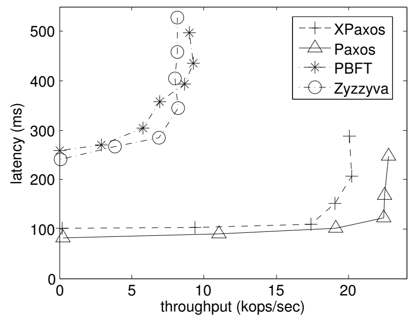
Table 4: Configurations of replicas. Shaded replicas are not used in the common case.



(a) 1/0 benchmark, $t = 1$



(b) 4/0 benchmark, $t = 1$



(c) 1/0 benchmark, $t = 2$

Figure 5: Fault-free performance

As we can see, in both benchmarks, XPaxos achieves a significantly better performance than PBFT and Zyzzyva. This is because, in a worldwide cloud environment, the network is the bottleneck and the message patterns of BFT protocols, namely PBFT and Zyzzyva, tend to be expensive. Compared with PBFT, the simpler message pattern of XPaxos allows better throughput. Compared with Zyzzyva, XPaxos puts less stress on the leader and replicates requests in the common case across 3 times fewer replicas than Zyzzyva (i.e., across t followers vs. across all other $3t$ replicas). Moreover, the performance of XPaxos is very close to that of Paxos. Both Paxos and XPaxos implement a round-trip across two replicas when $t = 1$, which renders them very efficient.

Next, to assess the fault scalability of XPaxos, we ran the 1/0 micro-benchmark in configurations that tolerate two faults ($t = 2$). We use the following EC2 datacenters for this experiment: CA (California), OR (Oregon), VA (Virginia), JP (Tokyo), EU (Ireland), AU (Sydney) and SG (Singapore). We place Paxos and XPaxos active replicas in the first $t+1$ datacenters, and their passive replicas in the next t datacenters. PBFT uses the first $2t+1$ datacenters for active replicas and the last t for passive replicas. Finally, Zyzzyva uses all replicas as active replicas.

We observe that XPaxos again clearly outperforms PBFT and Zyzzyva and achieves a performance very close to that of Paxos. Moreover, unlike PBFT and Zyzzyva, Paxos and XPaxos only suffer a moderate performance decrease with respect to the $t = 1$ case.

5.3 Performance under faults

In this section, we analyze the behavior of XPaxos under faults. We run the 1/0 micro-benchmark on three replicas (CA, VA, JP) to tolerate one fault (see also Table 4). The experiment starts with CA and VA as active replicas, and with 2500 clients in CA. At time 180 sec, we crash the follower, VA. At time 300 sec, we crash the CA replica. At time 420 sec, we crash the third replica, JP. Each replica recovers 20 sec after having crashed. Moreover, the timeout 2Δ (used during state transfer in view change, Section 4.3) is set to 2.5 sec (see Section 5.1.1). We show the throughput of XPaxos in function of time in Figure 6, which also indicates the active replicas for each view. We observe that after each crash, the system performs a view change that lasts less than 10 sec, which is very reasonable in a geo-distributed setting. This fast execution of the view-change subprotocol is a consequence of lazy replication in XPaxos that keeps passive replicas updated. We also observe that the throughput of XPaxos changes with the views. This is because the latencies between the primary and the follower and between the primary and clients vary from view to view.

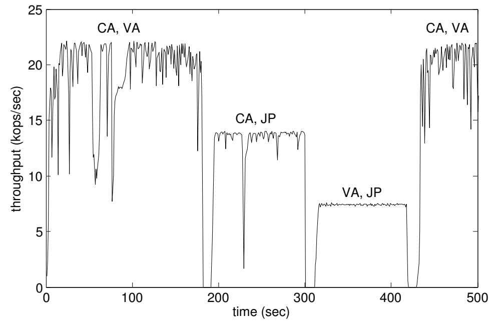


Figure 6: XPaxos under faults.

5.4 Macro-benchmark: ZooKeeper

To assess the impact of our work on real-life applications, we measured the performance achieved when replicating the ZooKeeper coordination service [19] using all protocols considered in this study: Zyzzyva, PBFT, Paxos and XPaxos. We also compare with the native ZooKeeper performance, when the system is replicated using the built-in *Zab* protocol [20]. This protocol is crash-resilient and requires $2t+1$ replicas to tolerate t faults.

We used the ZooKeeper 3.4.6 codebase. The integration of the various protocols inside ZooKeeper was carried out by replacing the Zab protocol. For fair comparison to native ZooKeeper, we made a minor modification to native ZooKeeper to force it to use (and keep) a given node as primary. To focus the comparison on the performance of replication protocols, and avoid hitting other system bottlenecks (such as storage I/O that is not very efficient in virtualized cloud environments), we store ZooKeeper data and log directories on a volatile *tmpfs* file system. The configuration tested tolerates one fault ($t = 1$). ZooKeeper clients were located in the same region as the primary (CA). Each client invokes 1 kB write operations in a closed loop.

Figure 7 depicts the results. The X-axis represents the throughput in kops/sec. The Y-axis represents the latency in ms. In this macro-benchmark, we find that Paxos and XPaxos clearly outperform BFT protocols and that XPaxos achieves a performance close to that of Paxos. More surprisingly, we can see that XPaxos is more efficient than the built-in Zab protocol, although the latter only tolerates crash faults. For both protocols, the bottleneck in the WAN setting is the bandwidth at the leader, but the leader in Zab sends requests to all other $2t$ replicas whereas the XPaxos leader sends requests only to t followers, which yields a higher peak throughput for XPaxos.

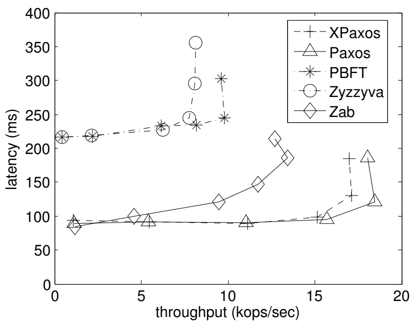


Figure 7: Latency vs. throughput for the ZooKeeper application ($t = 1$).

6 Reliability Analysis

In this section, we illustrate the reliability guarantees of XPaxos by analytically comparing them with those of the state-of-the-art asynchronous CFT and BFT protocols. For simplicity of the analysis, we consider the fault states of the machines to be independent and identically distributed random variables.

We denote the probability that a replica is correct (resp., crash-faulty) by $p_{correct}$ (resp., p_{crash}). The probability that a replica is benign is $p_{benign} = p_{correct} + p_{crash}$. Hence, a replica is non-crash-faulty with probability $p_{non\text{-}crash} = 1 - p_{benign}$. Besides, we assume there is a probability $p_{synchrony}$ that a replica is not partitioned, where $p_{synchrony}$ is a function of Δ, the network, and the system environment. Finally, the probability that a replica is partitioned equals $1 - p_{synchrony}$.

Aligned with the industry practice, we measure the reliability guarantees and coverage of fault scenarios using *nines of reliability*. Specifically, we distinguish *nines of consistency* and *nines of availability* and use these measures to compare different fault models. We introduce a function $9of(p)$ that turns a probability p into the corresponding number of "nines", by letting $9of(p) = \lfloor -\log_{10}(1-p) \rfloor$. For example, $9of(0.999) = 3$. For brevity, 9_{benign} stands for $9of(p_{benign})$, and so on, for other probabilities of interest.

Here, we focus on comparing *consistency* guarantees, which is less obvious than comparing availability, given that XPaxos clearly guarantees better availability than any asynchronous CFT or BFT protocol (see Table 1). The availability analysis can be found in [31].

6.1 XPaxos vs. CFT

We start with the number of *nines of consistency* for an asynchronous CFT protocol, denoted by $9ofC(CFT) = 9of(P[\text{CFT is consistent}])$. As $P[\text{CFT is consistent}] =$

p^n_{benign}, a straightforward calculation yields:

$$9ofC(CFT) = \left\lfloor -\log_{10}(1-p_{benign}) - \log_{10}(\sum_{i=0}^{n-1} p^i_{benign}) \right\rfloor,$$

which gives $9ofC(CFT) \approx 9_{benign} - \lceil \log_{10}(n) \rceil$ for values of p_{benign} close to 1, when p^i_{benign} decreases slowly. As a rule of thumb, for small values of n, i.e., $n < 10$, we have $9ofC(CFT) \approx 9_{benign} - 1$.

In other words, in typical configurations, where few faults are tolerated [13], a CFT system as a whole loses one nine of consistency from the likelihood that a single replica is benign.

We now quantify the advantage of XPaxos over asynchronous CFT. From Table 1, if there is no non-crash fault, or there are no more than t faults (machine faults or network faults), XPaxos is consistent, i.e.,

$$P[\text{XPaxos is consistent}] = p^n_{benign} + \sum_{i=1}^{t=\lfloor\frac{n-1}{2}\rfloor} \binom{n}{i} p^i_{non\text{-}crash}$$

$$\times \sum_{j=0}^{t-i} \binom{n-i}{j} p^j_{crash} \times p^{n-i-j}_{correct} \times \sum_{k=0}^{t-i-j} \binom{n-i-j}{k} \times$$

$$p^{n-i-j-k}_{synchrony} \times (1-p_{synchrony})^k.$$

To quantify the difference between XPaxos and CFT more tangibly, we calculated $9ofC(\text{XPaxos})$ and $9ofC(CFT)$ for all values of 9_{benign}, $9_{correct}$ and $9_{synchrony}$ ($9_{benign} \geq 9_{correct}$) between 1 and 20 in the special cases where $t = 1$ and $t = 2$, which are the most relevant cases in practice. For $t = 1$, we observed the following relation (the $t = 2$ case is given in [31]):

$$9ofC(\text{XPaxos}_{t=1}) - 9ofC(CFT_{t=1}) =$$

$$\begin{cases} 9_{correct} - 1, & 9_{benign} > 9_{synchrony} \wedge \\ & 9_{synchrony} = 9_{correct}, \\ min(9_{synchrony}, 9_{correct}), & \text{otherwise.} \end{cases}$$

Hence, for $t = 1$, we observe that the number of nines of consistency XPaxos adds on top of CFT is proportional to the nines of probability for a correct or synchronous machine. The added nines are not directly related to p_{benign}, although $p_{benign} \geq p_{correct}$ must hold.

Example 1. When $p_{benign} = 0.9999$ and $p_{correct} = p_{synchrony} = 0.999$, we have $p_{non\text{-}crash} = 0.0001$ and $p_{crash} = 0.0009$. In this example, $9 \times p_{non\text{-}crash} = p_{crash}$, i.e., if a machine suffers a faults 10 times, then one of these is a non-crash fault and the rest are crash faults. In this case, $9ofC(CFT_{t=1}) = 9_{benign} - 1 = 3$, whereas $9ofC(\text{XPaxos}_{t=1}) - 9ofC(CFT_{t=1}) = 9_{correct} - 1 = 2$, i.e., $9ofC(\text{XPaxos}_{t=1}) = 5$. XPaxos adds 2 nines of consistency on top of CFT and achieves 5 nines of consistency in total.

Example 2. In a slightly different example, let $p_{benign} = p_{synchrony} = 0.9999$ and $p_{correct} = 0.999$, i.e., the network behaves more reliably than in Example 1. $9ofC(CFT_{t=1}) = 9_{benign} - 1 = 3$, whereas $9ofC(\text{XPaxos}_{t=1}) - 9ofC(CFT_{t=1}) = p_{correct} = 3$, i.e., $9ofC(\text{XPaxos}_{t=1}) = 6$. XPaxos adds 3 nines of consistency on top of CFT and achieves 6 nines of consistency in total.

6.2 XPaxos vs. BFT

Recall that (see Table 1) SMR in asynchronous BFT model is consistent whenever no more than one-third of the machines are non-crash-faulty. Hence,

$$P[\text{BFT is consistent}] = \sum_{i=0}^{t=\lfloor\frac{n-1}{3}\rfloor} \binom{n}{i} (1-p_{benign})^i \times p^{n-i}_{benign}.$$

We first examine the conditions under which XPaxos has stronger consistency guarantees than BFT. Fixing the value t of tolerated faults, we observe that $P[\text{XPaxos is consistent}] > P[\text{BFT is consistent}]$ is equivalent to

$$p^{2t+1}_{benign} + \sum_{i=1}^{t} \binom{2t+1}{i} p^i_{non\text{-}crash} \times \sum_{j=0}^{t-i} \binom{2t+1-i}{j} p^j_{crash} \times$$

$$p^{2t+1-i-j}_{correct} \times \sum_{k=0}^{t-i-j} \binom{2t+1-i-j}{k} p^{2t+1-i-j-k}_{synchrony} \times$$

$$(1-p_{synchrony})^k > \sum_{i=0}^{t} \binom{3t+1}{i} p^{3t+1-i}_{benign} (1-p_{benign})^i.$$

In the special case when $t = 1$, the above inequality simplifies to

$$p_{correct} \times p_{synchrony} > p^{1.5}_{benign}.$$

Hence, for $t = 1$, XPaxos has *stronger consistency guarantees* than *any* asynchronous BFT protocol whenever the probability that a machine is correct and not partitioned is larger than the power 1.5 of the probability that a machine is benign. This is despite the fact that BFT is more expensive than XPaxos as $t = 1$ implies 4 replicas for BFT and only 3 for XPaxos.

In terms of nines of consistency, again for $t = 1$ ($t = 2$ is again given in [31]), we calculated the difference in consistency between XPaxos and BFT SMR, for all values of 9_{benign}, $9_{correct}$ and $9_{synchrony}$ ranging between 1 and 20, and observed the following relation:

$$9ofC(BFT_{t=1}) - 9ofC(\text{XPaxos}_{t=1}) =$$

$$\begin{cases} 9_{benign} - 9_{correct} + 1, & 9_{benign} > 9_{synchrony} \wedge \\ & 9_{synchrony} = 9_{correct}, \\ 9_{benign} - min(9_{correct}, 9_{synchrony}), & \text{otherwise.} \end{cases}$$

Notice that in cases where XPaxos guarantees better consistency than BFT ($p_{correct} \times p_{synchrony} > p_{benign}^{1.5}$), it is only "slightly" better and does not yield additional nines.

Example 1 (cont'd.). Building upon our example, $p_{benign} = 0.9999$ and $p_{synchrony} = p_{correct} = 0.999$, we have $9ofC(BFT_{t=1}) - 9ofC(\mathsf{XPaxos}_{t=1}) = 9_{benign} - 9_{synchrony} + 1 = 2$, i.e., $9ofC(\mathsf{XPaxos}_{t=1}) = 5$ and $9ofC(BFT_{t=1}) = 7$. BFT brings 2 nines of consistency on top of XPaxos.

Example 2 (cont'd.). When $p_{benign} = p_{synchrony} = 0.9999$ and $p_{correct} = 0.999$, we have $9ofC(BFT_{t=1}) - 9ofC(\mathsf{XPaxos}_{t=1}) = 1$, i.e., $9ofC(\mathsf{XPaxos}_{t=1}) = 6$ and $9ofC(BFT_{t=1}) = 7$. XPaxos has one nine of consistency less than BFT (albeit the only 7th).

7 Related work and concluding remarks

In this paper, we introduced XFT, a novel fault-tolerance model that allows the design of efficient protocols that tolerate non-crash faults. We demonstrated XFT through XPaxos, a novel state-machine replication protocol that features many more nines of reliability than the best crash-fault-tolerant (CFT) protocols with roughly the same communication complexity, performance and resource cost. Namely, XPaxos uses $2t+1$ replicas and provides all the reliability guarantees of CFT, but is also capable of tolerating non-crash faults, as long as a majority of XPaxos replicas are correct and can communicate synchronously among each other.

As XFT is entirely realized in software, it is fundamentally different from an established approach that relies on trusted hardware for reducing the resource cost of BFT to $2t+1$ replicas only [15, 30, 21, 39].

XPaxos is also different from PASC [14], which makes CFT protocols tolerate a subset of Byzantine faults using ASC-hardening. ASC-hardening modifies an application by keeping two copies of the state at each replica. It then tolerates Byzantine faults under the "fault diversity" assumption, i.e., that a fault will not corrupt both copies of the state in the same way. Unlike XPaxos, PASC does not tolerate Byzantine faults that affect the entire replica (e.g., both state copies).

In this paper, we did not explore the impact on varying the number of tolerated faults *per fault class*. In short, this approach, known as the *hybrid* fault model and introduced in [38] distinguishes the threshold of non-crash faults (say b) despite which safety should be ensured, from the threshold t of faults (of any class) despite which the availability should be ensured (where often $b \leq t$). The hybrid fault model and its refinements [11, 35] appear orthogonal to our XFT approach.

Specifically, Visigoth Fault Tolerance (VFT) [35] is a recent refinement of the hybrid fault model. Besides having different thresholds for non-crash and crash faults, VFT also refines the space between network synchrony and asynchrony by defining the threshold of network faults that a VFT protocol can tolerate. VFT is, however, different from XFT in that it fixes separate fault thresholds for non-crash and network faults. This difference is fundamental rather than notational, as XFT cannot be expressed by choosing specific values of VFT thresholds. For instance, XPaxos can tolerate, with $2t+1$ replicas, t partitioned replicas, t non-crash faults and t crash faults, albeit not simultaneously. Specifying such requirements in VFT would yield at least $3t+1$ replicas. In addition, VFT protocols have more complex communication patterns than XPaxos. That said, many of the VFT concepts remain orthogonal to XFT. It would be interesting to explore interactions between the hybrid fault model (including its refinements such as VFT) and XFT in the future.

Going beyond the research directions outlined above, this paper opens also other avenues for future work. For instance, many important distributed computing problems that build on SMR, such as distributed storage and blockchain, deserve a novel look at them through the XFT prism.

Acknowledgments

We thank Shan Lu and anonymous OSDI reviewers for their invaluable comments that helped us considerably to improve the paper. Shengyun Liu's work was supported in part by the National Key Research and Development Program (2016YFB1000101) and the China Scholarship Council. This work was also supported in part by the EU H2020 project SUPERCLOUD (grant No. 643964) and the Swiss Secretariat for Education, Research and Innovation (contract No. 15.0025).

References

[1] Crypto++ library 5.6.2. http://www.cryptopp.com/, 2014.

[2] B. Alpern and F. Schneider. Recognizing safety and liveness. *Distributed Computing*, 2(3):117–126, 1987.

[3] P.-L. Aublin, R. Guerraoui, N. Knežević, V. Quéma, and M. Vukolić. The next 700 BFT protocols. *ACM Trans. Comput. Syst.*, 32(4):12:1–12:45, Jan. 2015.

[4] P. Bailis and K. Kingsbury. The network is reliable. *Commun. ACM*, 57(9):48–55, 2014.

[5] J. Baker, C. Bond, J. C. Corbett, J. J. Furman, A. Khorlin, J. Larson, J. Leon, Y. Li, A. Lloyd, and V. Yushprakh. Megastore: Providing scalable, highly available storage for interactive services. In *Fifth Biennial Conference on Innovative Data Systems Research (CIDR)*, pages 223–234, 2011.

[6] P. Berman, J. A. Garay, and K. J. Perry. Towards optimal distributed consensus. In *Proc. 30th IEEE Symposium on Foundations of Computer Science (FOCS)*, pages 410–415, 1989.

[7] G. Bracha and S. Toueg. Asynchronous consensus and broadcast protocols. *J. ACM*, 32(4):824–840, 1985.

[8] B. Calder, J. Wang, A. Ogus, 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*, SOSP '11, pages 143–157. ACM, 2011.

[9] M. Castro and B. Liskov. Practical Byzantine fault tolerance and proactive recovery. *ACM Trans. Comput. Syst.*, 20(4):398–461, Nov. 2002.

[10] T. D. Chandra, R. Griesemer, and J. Redstone. Paxos made live: An engineering perspective. In *Proceedings of the Twenty-Sixth Annual ACM Symposium on Principles of Distributed Computing, PODC 2007*, pages 398–407, 2007.

[11] A. Clement, M. Kapritsos, S. Lee, Y. Wang, L. Alvisi, M. Dahlin, and T. Riche. Upright cluster services. In *Proceedings of the 22nd ACM Symposium on Operating Systems Principles, SOSP'09*, pages 277–290, 2009.

[12] A. Clement, E. L. Wong, L. Alvisi, M. Dahlin, and M. Marchetti. Making byzantine fault tolerant systems tolerate byzantine faults. In *Proceedings of the 6th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2009*, pages 153–168, 2009.

[13] J. C. Corbett, J. Dean, M. Epstein, et al. Spanner: Google's globally-distributed database. In *Proceedings of the 10th USENIX conference on Operating Systems Design and Implementation*, OSDI'12, pages 251–264, Berkeley, CA, USA, 2012. USENIX Association.

[14] M. Correia, D. G. Ferro, F. P. Junqueira, and M. Serafini. Practical hardening of crash-tolerant systems. In *2012 USENIX Annual Technical Conference*, pages 453–466, 2012.

[15] M. Correia, N. F. Neves, and P. Verissimo. How to tolerate half less one Byzantine nodes in practical distributed systems. In *Proceedings of the 23rd IEEE International Symposium on Reliable Distributed Systems*, SRDS '04, pages 174–183. IEEE Computer Society, 2004.

[16] F. Cristian, H. Aghili, R. Strong, and D. Dolev. Atomic broadcast: From simple message diffusion to Byzantine agreement. *Information and Computation*, 118(1):158–179, 1995.

[17] D. Dolev and H. R. Strong. Authenticated algorithms for Byzantine agreement. *SIAM Journal on Computing*, 12(4):656–666, Nov. 1983.

[18] C. Dwork, N. Lynch, and L. Stockmeyer. Consensus in the presence of partial synchrony. *J. ACM*, 35, April 1988.

[19] P. Hunt, M. Konar, F. P. Junqueira, and B. Reed. ZooKeeper: Wait-free coordination for internet-scale systems. In *2010 USENIX Annual Technical Conference*, pages 11–11, 2010.

[20] F. P. Junqueira, B. C. Reed, and M. Serafini. Zab: High-performance broadcast for primary-backup systems. In *Proceedings of the Conference on Dependable Systems and Networks (DSN)*, pages 245–256, 2011.

[21] R. Kapitza, J. Behl, C. Cachin, T. Distler, S. Kuhnle, S. V. Mohammadi, W. Schröder-Preikschat, and K. Stengel. CheapBFT: Resource-efficient Byzantine fault tolerance. In *Proceedings of the 7th ACM European Conference on Computer Systems*, EuroSys '12, pages 295–308, New York, NY, USA, 2012. ACM.

[22] R. Kotla, L. Alvisi, M. Dahlin, A. Clement, and E. Wong. Zyzzyva: Speculative Byzantine fault tolerance. *ACM Trans. Comput. Syst.*, 27(4):7:1–7:39, Jan. 2010.

[23] T. Kraska, G. Pang, M. J. Franklin, S. Madden, and A. Fekete. MDCC: multi-data center consistency. In *Eighth Eurosys Conference 2013*, pages 113–126, 2013.

[24] K. Krishnan. Weathering the unexpected. *Commun. ACM*, 55:48–52, Nov. 2012.

[25] P. Kuznetsov and R. Rodrigues. BFTW3: Why? When? Where? Workshop on the theory and practice of Byzantine fault tolerance. *SIGACT News*, 40(4):82–86, Jan. 2010.

[26] R. Ladin, B. Liskov, L. Shrira, and S. Ghemawat. Providing high availability using lazy replication. *ACM Trans. Comput. Syst.*, 10(4):360–391, Nov. 1992.

[27] L. Lamport. The part-time parliament. *ACM Trans. Comput. Syst.*, 16:133–169, May 1998.

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

[29] L. Lamport, R. Shostak, and M. Pease. The Byzantine generals problem. *ACM Trans. Program. Lang. Syst.*, 4:382–401, July 1982.

[30] D. Levin, J. R. Douceur, J. R. Lorch, and T. Moscibroda. TrInc: Small trusted hardware for large distributed systems. In *Proceedings of the 6th USENIX Symposium on Networked Systems Design and Implementation*, NSDI'09, pages 1–14. USENIX Association, 2009.

[31] S. Liu, P. Viotti, C. Cachin, V. Quéma, and M. Vukolić. XFT: Practical fault tolerance beyond crashes. *CoRR*, abs/1502.05831, 2015.

[32] S. Nakamoto. Bitcoin: A peer-to-peer electronic cash system. May 2009.

[33] B. M. Oki and B. H. Liskov. Viewstamped replication: A new primary copy method to support highly-available distributed systems. In *Proceedings of the Seventh Annual ACM Symposium on Principles of Distributed Computing*, PODC '88, pages 8–17, New York, NY, USA, 1988. ACM.

[34] D. Ongaro and J. K. Ousterhout. In search of an understandable consensus algorithm. In *Proc. USENIX Annual Technical Conference*, pages 305–319, 2014.

[35] D. Porto, J. a. Leitão, C. Li, A. Clement, A. Kate, F. Junqueira, and R. Rodrigues. Visigoth fault tolerance. In *Proceedings of the Tenth European Conference on Computer Systems*, EuroSys '15, pages 8:1–8:14, New York, NY, USA, 2015. ACM.

[36] F. B. Schneider. Implementing fault-tolerant services using the state machine approach: A tutorial. *ACM Comput. Surv.*, 22(4):299–319, 1990.

[37] Y. Sovran, R. Power, M. K. Aguilera, and J. Li. Transactional storage for geo-replicated systems. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, SOSP '11, pages 385–400, New York, NY, USA, 2011. ACM.

[38] P. M. Thambidurai and Y. Park. Interactive consistency with multiple failure modes. In *Proceedings of the Seventh Symposium on Reliable Distributed Systems, SRDS*, pages 93–100, 1988.

[39] G. S. Veronese, M. Correia, A. N. Bessani, L. C. Lung, and P. Veríssimo. Efficient Byzantine fault-tolerance. *IEEE Trans. Computers*, 62(1):16–30, 2013.

[40] M. Vukolić. The quest for scalable blockchain fabric: Proof-of-work vs. BFT replication. In *Open Problems in Network Security - IFIP WG 11.4 International Workshop, iNetSec 2015*, pages 112–125, 2015.

Realizing the fault-tolerance promise of cloud storage using locks with intent

Srinath Setty Chunzhi Su* Jacob R. Lorch Lidong Zhou
Hao Chen* Parveen Patel Jinglei Ren

Microsoft Research

Abstract

Cloud computing promises easy development and deployment of large-scale, fault tolerant, and highly available applications. Cloud storage services are a key enabler of this, because they provide reliability, availability, and fault tolerance via internal mechanisms that developers need not reason about. Despite this, challenges remain for distributed cloud applications developers. They still need to make their code robust against failures of the machines running the code, and to reason about concurrent access to cloud storage by multiple machines.

We address this problem with a new abstraction, called *locks with intent*, which we implement in a client library called *Olive*. Olive makes minimal assumptions about the underlying cloud storage, enabling it to operate on a variety of platforms including Amazon DynamoDB and Microsoft Azure Storage. Leveraging the underlying cloud storage, Olive's locks with intent offer strong *exactly-once* semantics for a snippet of code despite failures and concurrent duplicate executions.

To ensure exactly-once semantics, Olive incurs the unavoidable overhead of additional logging writes. However, by decoupling isolation from atomicity, it supports consistency levels ranging from eventual to transactional. This flexibility allows applications to avoid costly transactional mechanisms when weaker semantics suffice. We apply Olive's locks with intent to build several advanced storage functionalities, including snapshots, transactions via optimistic concurrency control, secondary indices, and live table re-partitioning. Our experience demonstrates that Olive eases the burden of creating correct, fault-tolerant distributed cloud applications.

1 Introduction

Cloud platforms such as Amazon AWS, Google Cloud, and Microsoft Azure are becoming popular choices for deploying applications because they permit elastic scaling, handle various operational aspects, and offer high reliability and availability. As a common practice, cloud platforms offer reliable storage services with simple APIs that hide the distributed nature of the underlying storage. Application developers are thereby freed from handling distributed-systems issues such as data partitioning, fault tolerance, and load balancing. Examples include Amazon's DynamoDB [1], Google's Cloud Storage [8], and Microsoft's Azure Storage [3]. This has led to a new paradigm for architecting applications where compute and storage components of an application are separated: applications store data on cloud storage, and perform computation on a set of client virtual machines (VMs).

This emerging architecture for applications poses an interesting new problem: Although cloud storage is made reliable by cloud service providers via fault-tolerance protocols [38, 49], it does not completely solve the problem of maintaining application-level consistency in face of failures. After all, clients running an application can fail, application processes on those clients can crash, and the network connecting those clients to the underlying storage can drop or reorder messages. Such issues can potentially leave the underlying storage in an inconsistent state, or block progress of application processes on other clients.

This problem is made even more challenging by the fact that cloud storage services tend to offer limited, low-level APIs. For example, the Azure Table storage service allows atomic batch update only on objects in the same partition [7, 21]. Cloud providers offer such APIs to allow efficient storage implementations, to offer applications the freedom to choose the right balance between performance and consistency, and to help themselves internally manage complexity and operational challenges. However, a limited API makes it hard for programmers of cloud applications to reason about correctness, given that clients can issue concurrent storage operations and can fail.

We address this problem with a new abstraction called *locks with intent*. The key insight behind this abstraction is that much of the complexity in handling failures and concurrency can be encapsulated in a simple *intent* concept that can be used in conjunction with locks. An intent is an arbitrary snippet of code that can contain both cloud storage operations and local computation, but with a key property that, when an intent execution completes, each step in the intent is guaranteed to have executed *exactly once*, despite failures, recovery, or concurrent executions.

A lock with intent lets a client lock an object in cloud storage as long as it first provides an intent describing what it plans to do while holding the lock. Once locked, the intent gains exclusive access to the object, just as

*Work done during an internship at Microsoft Research. Chunzhi Su is affiliated with The University of Texas at Austin, and Hao Chen is affiliated with Shanghai Jiao Tong University.

a traditional lock in a shared memory model. However, unlike a traditional lock, a locked object will *eventually* be unlocked even if the client holding the lock crashes, as long as the application is deadlock-free. Furthermore, before the lock is unlocked, each step in the associated intent is guaranteed to have been executed exactly once.

We implement this abstraction in a client library called Olive. Olive's design and implementation makes minimal assumptions about the underlying storage, which it encapsulates in the form of a common storage model. This model fits many existing cloud storage services as well as other large-scale distributed storage systems such as Apache Cassandra and MongoDB. Thus, Olive can work with any such storage service unchanged by using a shim that translates the service's API to the model's API.

To provide exactly-once execution semantics, Olive leverages the underlying storage's fault-tolerance properties. It stores each intent, with a unique identifier, in the underlying cloud storage system itself. Olive further introduces *distributed atomic affinity logging* (DAAL). DAAL colocates the log entry that corresponds to executing an intent step with the object changed by that step.

Olive also includes mechanisms to ensure progress. Because Olive provides exactly-once semantics even if multiple clients concurrently execute the same intent, any client can acquire any locked object by executing the associated intent. To ensure liveness for all intents, not just those associated with locks other clients wish to acquire, Olive introduces a special process called an *intent collector* that periodically completes unfinished intents.

Using Olive's locks with intent, we implement several libraries that provide advanced features on top of cloud storage. These include consistent snapshots, live table repartitioning, secondary indices, and ACID transactions. Our experience with these case studies suggests that Olive significantly reduces the burden on programmers tasked with making code robust to failures and concurrency. Furthermore, Olive's well-defined semantics make it easy to reason about correctness of application code despite failures and concurrency (§4, §5).

Our work makes the following contributions:

- We propose locks with intent, a new abstraction to simplify handling failures and concurrency in cloud applications built atop cloud storage services.
- We introduce a novel logging scheme called distributed atomic affinity logging (DAAL), and the idea of an intent collector. Together, they ensure exactly-once semantics despite failures and/or multiple clients executing the same intent.
- We demonstrate the feasibility of locks with intent by implementing them in Olive and making Olive compatible with a variety of cloud storage services.
- We demonstrate the generality and usability of locks with intent by using them to build several useful libraries and reason about their correctness.
- We experimentally evaluate Olive on Microsoft's Azure Storage to determine the performance cost of using locks with intent compared to baselines providing similar fault-tolerance guarantees.

2 Building cloud applications: challenges

Cloud applications typically run on multiple client VMs and store state on cloud storage: the client VMs are used only for computation and are effectively stateless. Such applications are fundamentally distributed and must cope with distributed-systems challenges such as asynchrony, concurrency, failure, and scaling.

The underlying reliable distributed cloud storage aims to alleviate the difficulty of building cloud applications. Its API thus generally hides the complexity of concurrency control, elasticity, and fault tolerance. Nevertheless, the developer of a cloud application still has to handle VM failures. She must also bridge the gap between rich application semantics and the cloud storage's simple API.

2.1 A common storage model

Different cloud storage services offer different, constantly-evolving APIs. But, we want Olive to operate on any cloud service without requiring significant reworking each time a provider decides to make changes. Thus, we introduce a *common storage model*, an API that has enough features to support Olive but is simple enough to be implemented by any cloud storage service. In particular, it is easily implemented by popular cloud storage systems such as Microsoft's Azure tables and Amazon's DynamoDB, and by large-scale distributed storage systems such as Apache Cassandra and MongoDB. By stripping away functionality unique to certain services and focusing only on basic operations, we enable broad applicability for Olive.

Our model is that of a storage system providing schemaless tables. Each table row, also called an *object*, consists of a key and a set of attribute/value pairs. A table may be divided into *partitions* to satisfy a system-imposed limit on maximum partition size.

API. The model's API includes operations to `Create`, `Read`, `Update`, and `Delete` rows (CRUD). It also includes `Scan`, `UpdateIfUnchanged`, and `AtomicBatchUpdate`, described in the next paragraphs.

`Scan` takes a table and a predicate as parameters, and returns a stream providing all rows in that table satisfying that predicate. For instance, the predicate might be "has a count attribute with value > 5." Every row that satisfies the predicate throughout the scan is guaranteed to be included. A row that only satisfies the condition some time during the scan (e.g., because it was created, updated, or deleted during the scan) may or may not be included.

`UpdateIfUnchanged` is like `Update`, except it does nothing if the object to be updated has been updated or deleted since a certain previous operation on that object. That previous operation is identified by a *handle* passed to `UpdateIfUnchanged`. The application can obtain such handles because each `Create`, `Read`, and `Update` operation returns a handle representing that operation.

`AtomicBatchUpdate` lets the application perform multiple update and insert operations atomically. In other words, despite possible failures, either all or none of the operations will happen. However, this atomicity guarantee only works at a certain granularity: objects passed to `AtomicBatchUpdate` must be in the same atomicity scope, where the scope is a system-specific parameter.

Such a storage model is supported not only by cloud storage services, such as Amazon DynamoDB (with rows as the atomicity scope) and Microsoft Azure table storage (with partitions as the atomicity scope), but also by popular storage systems, such as MongoDB (with documents as the atomicity scope) and Cassandra (with partitions as the atomicity scope). Azure Table supports ETags, which can be considered as handles; DynamoDB supports conditional update. MongoDB supports *Update if Current* and Cassandra supports the IF keyword in INSERT, UPDATE and DELETE statements for conditional updates, which can be considered as generalizations of the conditional update primitive in our model. Such common capabilities are chosen by different storage services because they provide simple and flexible primitives for concurrency-control and fault-tolerance support, and because they can be supported at a manageable cost and complexity. The cost and complexity consideration leads to a somewhat limited API. For example, Cassandra chooses to support partition-level atomicity because "the underlying Paxos implementation works at the granularity of the partition" [5].

Invisible entries. In our model, it is always possible to put *invisible* entries in a scope. That is, a library interposing on the API between the application and the cloud storage can put entries in a scope, but hide them from the application by stripping them from returned results. Even if the only scope available is an object, this can be done by adding special attributes to it. If scopes are larger, such as partitions or tables, the library can use special rows.

Invisible entries should be used sparingly since they reduce performance and capacity. They reduce performance when an access to a real entry necessitates one or more accesses to invisible entries. They reduce capacity by using space that could otherwise be used for application data. In particular, a cloud storage system often places an upper bound on the size of a scope, e.g., a maximum row or partition size. By using invisible entries, the interposing library reduces the effective maximum size from the application's perspective. Indeed, when the application asks

```
1  def updateObject(key, newObj):
2    obj = curTable.Read(key)
3    lastSnapshot = curTable.Read(LAST_SNAP).value
4    curEpoch = lastSnapshot + 1
5
6    if (obj != None and obj.version <= lastSnapshot):
7      snapshotTables[lastSnapshot].Update(key, obj)
8
9    newObj.version = curEpoch
10   curTable.UpdateIfUnchanged(key, newObj)
```

FIGURE 1—Pseudocode for the object-update routine in a buggy snapshot design. It sometimes requires extra work because a snapshot table is being lazily populated.

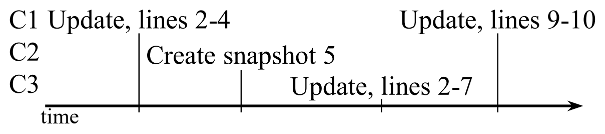


FIGURE 2—Execution trace exposing a bug in the Figure 1 code. Client C1, a slow updater, performs update lines 2–4, looking up a certain key and finding version #5. Since the last snapshot epoch is #4, the object is up to date and no copy-on-write is needed so it skips to line 9. Update lines 9–10, which happen later, update the current table with new contents but still version #5. Meanwhile, client C2 creates a new snapshot, and client C3 does an update involving a copy-on-write to snapshot table #5. At the end of this trace, the snapshot invariant is violated: snapshot table #5 contains contents that do not reflect the latest update, just performed by C1.

for the maximum allowable size of a scope, the library must provide a lower number than the underlying storage system to account for this.

Lock. The primitives provided by the storage interface can be used to implement other useful client functionality. For example, we can implement an object lock by adding an invisible Boolean attribute called `locked` to the object. To acquire the lock, a client reads the object and gets a handle. If the `locked` bit is not set, the client issues a conditional update with the returned handle to set the `locked` bit. It gets the lock if and only if that update succeeds. To release the lock, the client resets the `locked` bit via another update.

2.2 A case study: supporting snapshots

As a case study, we present the example of supporting storage snapshots using the common storage model described earlier. A resulting *table-snapshotting* (or STable) service allows clients to create snapshots of a table without interrupting normal operations on the table, in addition to the standard CRUD operations. This is functionality we have actually designed and implemented for a production scenario.

To demonstrate how easy it is to accidentally introduce bugs when designing snapshot tables, we show one of our earliest designs and the bug it contained. This design implements snapshotting tables directly on cloud storage, instead of using primitives like locks with intent.

Buggy STable design. In this design, each snapshot is implemented as an actual table. However, rather than fully populating this table when a snapshot is created, the table is populated lazily. This makes snapshot creation quick, which prevents snapshot creation from making the table unavailable for an extended period of time.

Snapshots are numbered in increasing order, with the first one being snapshot 1. Snapshots divide time into epochs, with epoch 1 preceding snapshot 1, epoch 2 coming between snapshots 1 and 2, etc. An invisible entry is put into each object to represent its *version*, defined as the last epoch it was updated in. An invisible entry is put into the table to represent the number of the last snapshot taken. The current epoch is one more than this number.

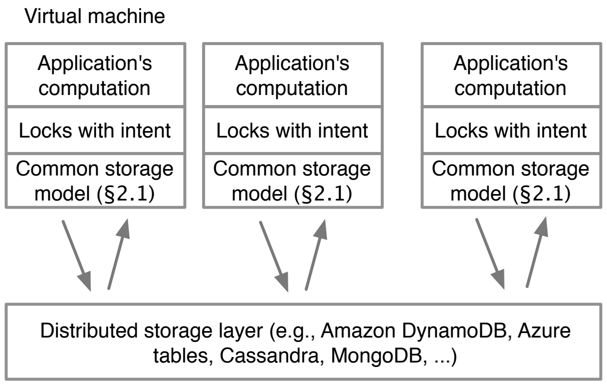
To lazily populate a snapshot, we use a snapshot-aware routine for updating objects, as shown in Figure 1. If it finds that the object version in the current table belongs in an earlier snapshot (i.e., smaller than the current snapshot number of the table), it copies the object to a snapshot table before overwriting it. This makes the current table essentially copy-on-write after a new snapshot is taken. A key *snapshot* invariant for STable is that, if snapshot table *i* has a row with key *k*, then that row contains the contents of the last update to key *k* made with version *i*.

Figure 2 illustrates an example execution demonstrating a bug in this design that violates the snapshot invariant. The subtle bug surfaces because a client holds on to an old snapshot number for the STable and completes its update only after a new snapshot is created and after another client performs a copy-on-write. This delayed update associates different object contents with the version copied to the snapshot, thereby violating the snapshot invariant. The use of conditional update does not help because copy-on-write is a multi-object operation.

One way to fix this bug is for the client to acquire a lock on the object for the duration of the code in Figure 1. This would prevent multiple updates from interleaving. While implementing a lock is feasible as shown earlier, one challenge is to ensure liveness when a lock holder fails. To relieve developers from worrying about these subtle issues and to help reason about correctness despite concurrency and failures, we introduce a new primitive called locks with intent, which the next section elaborates.

3 Locks with intent

As shown in the STable example of §2, the main challenge in developing cloud applications is to ensure correctness in the face of client failures and concurrent cross-scope client operations. Olive therefore introduces locks with intent, which ensures *exactly-once* execution (despite failures) and *mutual exclusion* (for concurrent operations).



Virtual machine
Application's computation
Locks with intent
Common storage model (§2.1)
Application's computation
Locks with intent
Common storage model (§2.1)
Application's computation
Locks with intent
Common storage model (§2.1)
Distributed storage layer (e.g., Amazon DynamoDB, Azure tables, Cassandra, MongoDB, ...)

FIGURE 3—Olive's high level architecture. Olive exposes the abstraction of locks with intent (§3) to higher-level application in the form of a library. The abstraction provides eventual exactly-once execution semantics despite failures of nodes running the Olive library. Olive provides such strong semantics by leveraging the fault-tolerance properties of the underlying distributed storage layer.

We intend Olive to be used by both application and infrastructure developers. Since our approach is flexible enough to support a transaction library, as we will show in §4.4, users can always use that library to have the same simplicity offered by transactions. But, crucially, our design also allows sophisticated users to write more efficient implementations, by reducing complexity via automatic failure handling and simplification of concurrency.

Figure 3 depicts Olive's high level architecture. Olive's locks with intent provide a new abstraction for cloud applications to handle failures and concurrency elegantly. This abstraction is built atop the common storage model described in §2.1, which can be mapped to different cloud storage or distributed storage systems. Olive does not modify the storage layer, so it preserves the performance and scalability characteristics of existing storage services. Furthermore, Olive does not require direct distributed coordination among clients running an application's computation: all interactions are through cloud storage, conforming to the existing cloud application model.

3.1 Intents: Exactly-once execution

An *intent* is a request for a certain code snippet to be executed exactly once. The snippet may contain loops or recursive calls, but must terminate in a bounded number of steps. An intent can involve both local computation and operations on cloud storage. The code snippet is arbitrary, but usually it is a critical section protected by a lock.

Determinism. Besides bounded run time, the main restriction on intent code is that it be deterministic. That is, it must produce the same result when executed with the same inputs and in the same state. Determinism makes it possible to replay an execution after a failure by precisely reproducing results up to the failure point and then continuing execution. Non-determinism is permitted only in Olive-provided routines, where Olive can track the sources of non-determinism and return the same result deterministically. For instance, Olive provides a routine for generating random numbers; the developer must use it instead of the system random number generator.

The code must be deterministic even if run by different clients. For instance, it should not depend on any special privileges possessed only by a subset of clients. §6 will discuss removing this restriction in future work.

Non-deterministic code in an intent constitutes a bug. Olive cannot detect this error; it simply does not guarantee exactly-once semantics in this case.

Tracking and executing intents. Exactly-once execution of an intent is challenging because (i) the initiating client may fail partway through executing it, and (ii) other clients attempting to recover from the initiator's failure may lead to multiple, possibly concurrent, clients executing the intent. After all, failure detection may be imperfect, so one client may incorrectly believe another has failed and attempt to recover from that apparent failure. Olive deals with these challenges as follows.

Olive assigns a unique `intentId` to each intent, and uses this as a key when storing the intent in the `intents` table. To ensure exactly-once execution semantics, Olive must log the steps any client executes as part of an intent. This way, if the client fails, another client will know where to continue from during recovery. Olive does this logging in a table named `executionLog`. For each step requiring logging, Olive adds a new row to `executionLog`, using a key combining `intentId` and the step number within the intent. For local non-deterministic operations, such as those done by Olive-provided random-number-generation routines, Olive stores any non-deterministic choices in `executionLog`. For cloud-storage operations that return results (e.g., reads), Olive stores those results in `executionLog`. This logging allows any future re-execution of an intent to return the same result.

Due to the limited storage model described in §2.1, Olive cannot *atomically* read an object from cloud storage and write it to `executionLog`. Fortunately, it does not have to, because read operations have no immediate externally visible effect. In fact, for better performance, Olive defers logging until right before it executes an externally visible operation. As a result, a client could crash immediately after issuing a read operation to the cloud storage, but before logging to `executionLog`. In that case, if a client resumes executing the intent, it will re-execute the read operation, potentially getting a different value from the cloud storage. This is safe because only the new execution leads to externally visible effects.

DAAL. We have so far treated `executionLog` as if it were a single standalone database table, but as we will now discuss it is only *logically* a single table. To achieve exactly-once semantics, when we update data we must also, in one atomic action, log that update to `executionLog`. But, as discussed in §2.1, most cloud storage services do not support atomic actions across tables. Thus, while it is possible to store logs of read operations in a single table, each log of a write operation must be in the same table as the object being written.

To solve this problem, Olive introduces a novel logging scheme called *distributed atomic affinity logging* (DAAL). With DAAL, `executionLog` consists of two parts. The first part, which stores the results of completed read operations, is a regular table. The second part, which stores log entries corresponding to writes, is a set of invisible entries distributed among the scopes in the system. To perform an `AtomicBatchUpdate` that both updates an object and inserts an `executionLog` entry, Olive chooses a scope for the entry that includes the updated object.

Olive deterministically derives each entry's identifier from the `intentId`, the current intent step number, and the key of the modified object. This ensures that, if another client later tries to perform the update a second time, it will fail because the invisible log entry already exists. If the log entry is a row, then the second insertion will fail because of a key conflict. If the log entry is an attribute, the second insertion will fail because the client will first do a read to ensure the absence of the attribute, then will perform the attribute insertion using `UpdateIfUnchanged`.

Olive also logs progress information in the central `intents` table—one column of each intent's row indicates how many update steps of that intent have been executed. The steps recorded in the intents table must actually have been performed although additional steps may have been executed that are not yet recorded. This is just an optimization to avoid clients wasting time attempting to re-perform already-executed steps. Not recording an already-executed step does not compromise correctness because DAAL ensures that no client will be able to successfully execute any update a second time.

Note that the first part of `executionLog`, the single table holding logs of read operations, is accessed by every client performing an intent. To prevent this table from becoming a throughput bottleneck, or from exceeding capacity limits, Olive partitions it on `intentId`. The degree of partitioning is configurable.

Liveness. Exactly-once semantics requires more than just never executing any intent more than once. It also requires executing each intent *at least* once. We ensure this liveness property as follows.

First, we put another requirement on intent code besides non-determinism and bounded run time. The developer must ensure that, as long as the code is retried repeatedly, it eventually completes.

Given this requirement, all Olive must do to achieve liveness is to retry each intent repeatedly. To ensure such repeated retries, Olive uses an *intent collector*. This special background process periodically scans the `intents` table to identify incomplete intents and complete them.

Such an intent collector guarantees liveness as long as it never stops. Fortunately, cloud providers offer mechanisms to monitor core services and to restart them if they fail; such a mechanism should be used for the intent collector. Even if this causes multiple instances of the intent collector to coexist briefly, this is safe because of Olive's assurance of at-most-once semantics for each intent.

Indeed, it may be desirable to *always* run multiple instances of the intent collector, so that if one fails and the failure takes time to be detected and rectified, intents are still completed promptly. Multiple instances may, for efficiency, be designed to partition work among themselves, but we have not yet implemented such partitioning.

3.2 Mutual exclusion with exactly-once semantics

Intents can be combined with locks to ensure both mutual exclusion and exactly-once semantics, leading to a powerful new primitive called locks with intent. A "lock" in this context is like a typical lock in that it restricts access to an object or set of objects. However, the access restriction is not to a single client but to a single intent: only clients performing that intent are permitted access. That intent has a step that acquires the lock, then steps that access the locked objects, then a step that releases the lock.

From the developer's perspective, the lock is easy to use since it acts like a regular lock that restricts object access to only a single client. In reality, locked objects are accessible to multiple clients, but, because of the exactly-once semantics of intents, all those clients' object accesses are equivalent to accesses by a single client. Thus, semantically, acquiring one of our locks is equivalent to acquiring a lock that limits access only to a single client.

Even though our locks are semantically equivalent to normal locks, they are safer to use. An object locked with a normal lock can only be accessed by the client who locks it. This is dangerous since the client may fail, rendering the object forever unavailable. However, by allowing any client performing the intent to access the locked object, the developer no longer has to worry about this concern. Locks cannot be tied up indefinitely; they will eventually be released by the intent collector.

Despite this, a client that needs to access an object may still have to wait a long time for the collector to release an intent lock on it. Thus, as an additional optimization, we introduce the following mechanism. When code within an intent acquires a lock, we associate the intent's `intentId` with that lock using an invisible attribute. When the code later releases the lock, we remove the association with the intent's `intentId`. This way, if another client needs the lock but finds it unavailable, it can tell whether the lock is held by an intent. If so, the blocked client can take responsibility for immediately completing the intent, thereby allowing itself to make progress.

```
def updateObject_IntentCode(key, newObj):
  obj = curTable.Read(key)
  if obj == None:
    return NOT_FOUND

  table.Lock(obj.key)
  lastSnapshot = curTable.Read(LAST_SNAP).value
  curEpoch = lastSnapshot + 1

  if (obj.version <= lastSnapshot):
    snapshotTables[lastSnapshot].Update(key, obj)

  newObj.version = curEpoch
  curTable.UpdateIfUnchanged(key, newObj)
  table.Unlock(obj.key)
  return SUCCESS
```

FIGURE 4—Pseudocode for the intent code to update an object.

4 Applications and experience

Locks with intent make it easy to reason about desirable correctness and fault-tolerance properties of software. To demonstrate their general utility, this section will describe how we use them to build several components.

Note that these components are themselves generally useful. That is, each is a library that provides applications with a storage API richer than that of the underlying cloud storage system. In §4.1, we discuss our STable library, which augments the cloud storage API with a facility for snapshotting tables. In §4.2, we discuss a library that adds the ability to do live table re-partitioning. In §4.3, we show how to add support for secondary indices. Finally, in §4.4, we show how to add the ability to form ACID transactions out of arbitrary sequences of operations.

4.1 Snapshots

One component we build is the STable library, which provides applications with the ability to take snapshots of tables. In §2.2, we discussed how the complexity of table snapshotting can lead to subtle bugs. In particular, the code in Figure 1 can lead to a violation of our snapshot invariant. The fundamental reason the bug arises is the difficulty of reasoning about the many possible interleavings of concurrent clients.

Fortunately, intents provide a straightforward way to reduce the possible interleavings. That is, we can create an intent that locks `obj` while executing the code from Figure 1; the resulting intent code is shown in Figure 4. Because the intent locks `obj`, executions of intents on the same obj are serialized; i.e., they do not overlap. Furthermore, we do not have to worry about liveness issues arising from introducing locks, because locks with intent automatically defend against failing lock holders.

Here is an argument that the snapshot invariant is maintained by this approach. Because the intent locks obj, all executions of the intent on the same obj are serialized, i.e., they do not overlap. Consider any run of the intent that copies the contents of obj to snapshot table i. Because this copy occurs in the middle of an intent, and all intents to obj are serialized, the copy must reflect all earlier executions of the intent. That is, it must reflect the last update performed so far, and the snapshot invariant holds. We must also demonstrate that the invariant continues to hold, i.e., that a later update will not violate it by writing to the current table with version i. To demonstrate this, we observe that any subsequent run of the intent for obj will be serialized afterward. Those runs will read a lastSnapshot $\geq i$, causing them to use a curEpoch $\geq i+1$. Thus, the snapshot invariant is maintained.

Note that the only object we lock is obj; we do not lock the special row with key LOCK_SNAP. Thus, we do not conflict with concurrent operations that update the current snapshot number. If we were to use transactions instead of locks with intent, we would have such a conflict.

Our STable implementation offers stronger properties than just the snapshot invariant. For instance, it ensures that any two reads of the same key from the same snapshot will return the same object contents. It also offers further functionality, like the ability to garbage-collect old snapshots and to roll back to earlier snapshots. These facilities also became easier to build with locks with intent.

4.2 Live table re-partitioning

Another component we build is a library that exports a facility for live re-partitioning of tables. This functionality is crucial if a table may grow to the point where it exceeds system-imposed size limits. It can also help relieve "hot spots" by dividing a frequently-accessed tables into multiple tables with consequently greater throughput. By building this library, we do for general cloud storage what Zephyr [28] did for transactional storage.

A straightforward approach would be to lock the table for the duration of re-partitioning. However, re-partitioning potentially involves an enormous amount of data movement, taking seconds or minutes. So, it is unreasonable to block clients during re-partitioning; we must allow concurrent operations during re-partitioning.

This concurrency requirement poses challenges for correct development. The developer must now reason about all the possible interleavings of client operations with steps of re-partitioning. Failure to do so can lead to bugs.

To illustrate this, Figure 5 depicts a buggy design aimed at enabling object updates during live re-partitioning of

```
1  def migratePartitionToNewTable(pKey, futTable):
2    curTable = metaTable.Read(pKey).value
3    metaTable.Update(pKey, [curTable, futTable])
4
5    objectsToMove =
6      Scan(curTable, partitionKey == pKey)
7    for (obj in ObjectsToMove):
8      futTable.Create(obj.key, obj)
9    metaTable.Update(pKey, [futTable])
10
11 def updateObject(key, newObj):
12   # get partition key associated with the key
13   pKey = getPartitionKey(key)
14   tablesList = metaTable.Read(pKey).value
15
16   # check if this table is being re-partitioned
17   if (tablesList.len == 1):
18     curTable = tablesList[0]
19     curTable.Update(key, newObj)
20   else: ...
```

FIGURE 5—Pseudocode for the object-update routine and a migration routine in a buggy live re-partitioning design.

tables. The bug arises in the following scenario. Suppose that, when a client starts executing updateObject for key k, there is no ongoing re-partitioning job, so the client reaches line 17. At this point, a re-partitioning job commences, and successfully migrates key k to the new partition. The client then continues from line 17, writing its update only to a table that will soon be obsolete. Eventually, the re-partitioning job reaches line 9 without realizing there is useful data it missed in the current table. So, when it updates the metaTable, it effectively and incorrectly rolls back the client's update.

Fortunately, such challenges and reasoning can be substantially mitigated due to locks with intent. Our general strategy is to break the job of re-partitioning into small tasks, each of which is short enough that it is acceptable to block clients for its duration. We then use a lock with intent for each such task, and a lock with intent for each client operation on the table.

In this way, we do not have to reason about arbitrary interleavings between clients and the re-partitioning job. We only have to reason about interleavings at the coarse scale of tasks. For instance, a client operation can overlap the re-partitioning job, but it cannot overlap a task that accesses the same object. More specifically, each task corresponds to migrating one object from one partition to another partition. This involves replacing the object in the old partition with a marker that redirects clients with outdated views to the new partition. Figure 6 depicts pseudocode for the migration routine as well the object-update procedure in the re-partitioning service that uses locks with intent.

With this migrator design, the object-update routine does not use locks. If an object is locked for migra-

```
def migrateIntent(curTable, futTable, obj):
  curTable.Lock(obj.key)
  futTable.Create(obj.key, obj)
  obj.migrated = True
  curTable.Update(obj.key, obj)
  curTable.Unlock(obj.key)

def migratePartitionToNewTable(pKey, futTable):
  curTable = metaTable.Read(pKey).value
  metaTable.Update(pKey, [curTable, futTable])
  objsToMove =
     Scan(curTable, partitionKey == pKey)
  for (obj in ObjsToMove):
    migrateIntent(curTable, futTable, obj)
  metaTable.Update(pKey, [futTable])

def updateObject(key, newObj):
  pKey = getPartitionKey(key)
  tablesList = metaTable.Read(pKey).value
  curTable = tablesList[0]
  if (tablesList.len == 1):
    curTable.UpdateIfUnchanged(key, newObj)
  elif (tablesList.len == 2):
    futTable = tablesList[1]
    oldObj = curTable.Read(key)
    if (oldObj.migrated == True):
      futTable.UpdateIfUnchanged(key, newObj)
    elif (oldObj.locked == True):
      migrateIntent(curTable, futTable, oldObj)
      futTable.UpdateIfUnchanged(key, newObj)
    else:
      curTable.UpdateIfUnchanged(key, newObj)
```

FIGURE 6—Pseudocode for the migration routine and the object-update routine in the live table re-partitioning service based on Olive's locks with intent.

tion, it assists the migrator by executing the associated intent, before performing its update. Otherwise, it uses `UpdateIfUnchanged` to modify the object in the old partition (if the object is not migrated or if no migration in progress), or in the new partition (if the object is already migrated). If a client holds an outdated view (e.g., it incorrectly thinks no migration is in progress or an object is not migrated), the `UpdateIfUnchanged` fails, causing it to retry, which will update its view. Furthermore, unlike in the buggy design shown earlier, it is safe for the object-update routine to update an object in the old partition as long as it has not been migrated because the migrator will eventually move the updated object to the new partition.

Of course, locks with intent are not a panacea. There are still several tricky cases to consider, such as how to avoid conflict between a re-partitioning task and a client with an outdated view attempting to insert an object with the same key. However, we find that the number of cases to consider is much smaller thanks to the coarsening of operations enabled by locks with intent.

4.3 Secondary indices

Another component we build is a library that supports constructing, maintaining, and using secondary indices. A secondary index for a table `T` is a separate table `T'` designed to allow quick lookups into `T` using a non-key attribute `Attr`. Each row of `T'` consists of an `Attr` value and a `T` key. However, `T'` uses the `Attr` values as its keys.

The main challenge in building secondary-index support is maintaining consistency between `T` and `T'`. Because cloud storage systems typically do not support multi-table atomic transactions, there are necessarily times when the two tables' contents are not consistent with each other. For example, a row may exist in `T` without a corresponding row in `T'`.

To see the challenge more concretely, consider the following naïve algorithm for updating an object in `T`:

1. Update the corresponding row in `T`.
2. Insert a row into `T'` mapping the new `Attr` value to the key of the updated row.
3. Delete the row from `T'` with the former `Attr` value.

Unfortunately, this logic is not robust to failures: if a process running the above procedure crashes after step 1 but before completing steps 2 and 3, it will leave the underlying storage in an inconsistent state.

We could address this with "cleanup" processes that run in the background to periodically find and fix inconsistencies between `T` and `T'`. However, in addition to complicating deployment and wasting resources by continuously scanning for inconsistencies, such cleanup processes can actually *introduce* inconsistency, as in the following scenario. First, a client updates an object to change `Attr` from OLD to NEW, but crashes before step 3. Next, a cleanup process notices this and decides to delete the row in `T'` with `Attr`=OLD. Next, another client decides to change `Attr` back to OLD, and completes steps 1 through 3. Finally, the cleanup process acts on its earlier decision and deletes the row in `T'` corresponding to OLD, not realizing that this is actually now a useful row. This leaves `T'` without any row corresponding to the object.

Olive provides a natural solution to eventually consistent secondary indices. We perform steps 1–3 described earlier in an intent that also locks the object from `T`. Because secondary indices are eventually consistent when the intents complete their executions, we do not have to worry about inconsistencies caused by intermediate failures, like the one discussed earlier. Additionally, the intent collector in this solution has to do less work than the cleaner process described earlier. After all, the cleaner process must scan *all* rows changed since the last time it ran, but the intent collector only has to scan the `intents` table for outstanding incomplete intents.

```
def atomicCommit(objectsRead, objectsModified):
  for (obj in objectsModified):
    table.Lock(obj.key)

  success = True
  for (obj in objectsRead): # verify read set
    retrievedObj = table.Read(obj.key)
    if (retrievedObj.version != obj.version):
      success = False
      break

  if (success): # commit and unlock
    for (obj in objectsModified):
      obj.locked = False
      table.Update(obj.key, obj)
  else: # abort and unlock
    for (obj in objectsModified):
      table.Unlock(obj.key)
```

FIGURE 7—Pseudocode for atomic commit in OCC-based transactions.

4.4 Transactions

The last component we build is a library that augments the cloud storage API with the ability to form transactions out of an arbitrary collection of operations. This is valuable because, as described in §2.1, most cloud storage systems do not support transactions across tables. We will see that Olive's locks with intent make it simple to build a client-side library that exports APIs to execute general-purpose ACID transactions [16, 33, 40, 55].

Our design of this transaction library is based on optimistic concurrency control (OCC), which has three phases: shadow execution, verification, and update in place [37]. To guarantee ACID semantics, an OCC protocol requires that the last two steps happen atomically. In particular, they must be isolated from updates of other transactions. Furthermore, all changes made by the transaction must be either committed or aborted in their entirety. Thus, a core piece of a distributed transaction protocol is the atomic-commit mechanism. To satisfy these requirements, distributed systems that implement transactions use the following techniques: a special *transaction coordinator* process uses a shadow write-ahead log to ensure atomicity, and uses locks during the verification step to ensure isolation from other transactions [27].

We observe that these techniques can be naturally implemented by using Olive's locks with intent. Figure 7 depicts pseudocode that can be wrapped in an intent to execute the atomic commit mechanism with the aforementioned properties. The mutual exclusion property of Olive's locks with intent provides the desirable isolation, and the intent's execution log acts as a write-ahead log. That is, it contains all the information needed to commit or abort a transaction. Most importantly, for liveness we require that transaction coordinators never fail; we ensure this by using intents as our transaction coordinators.

service	without Olive	with Olive
snapshots	987	665
OCC-transactions	2,201	408
live re-partitioning	2,116	474

FIGURE 8—Comparison of code line counts for services we built with and without Olive.

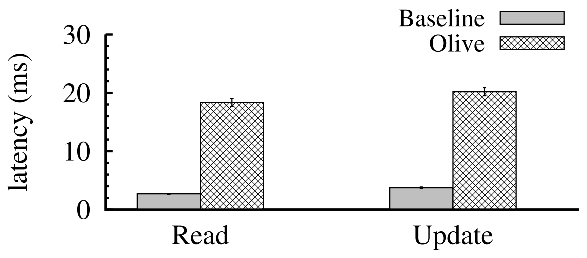


FIGURE 9—Latency of executing a storage operation, either inside an intent or directly on the raw storage interface. The logging required to ensure exactly-once execution semantics adds up to 6–7× the baseline latency. (See text for details.) Figure 10 depicts how these overheads are amortized when an intent contains more than one storage operation.

4.5 Evaluation: ease of development

Before we designed Olive's locks with intent, we created some of the cloud services described in this section by building directly atop the raw cloud-storage interface. It was both tedious and error-prone as we had to reason about many failure scenarios and consider many interleavings of code steps by different clients. To concretely demonstrate how Olive makes it easy to develop such cloud services, we now compare the complexity of developing such cloud services with and without Olive. We use lines of code as a proxy for code complexity.

Figure 8 depicts our results. These results demonstrate that Olive reduces lines of code written, and thus likely reduces complexity. This finding, in combination with our experience (§4.1–4.4), suggests that Olive makes it significantly easier to build these services. Note that one of the artifacts that does not use Olive (live re-partitioning) was built by a different team with a very different approach to making code robust to failures and concurrency. (We note this because the comparison and feature set may not be fully apples-to-apples.) For the case of OCC-based transactions, as discussed in §4.4, Olive makes it simple to express a transactional protocol.

5 Experimental evaluation

The previous section demonstrated that Olive's locks with intent make it easy to design, and to reason about the correctness of, new cloud services that are robust to failures and concurrency. This section experimentally evaluates Olive to understand its costs and benefits.

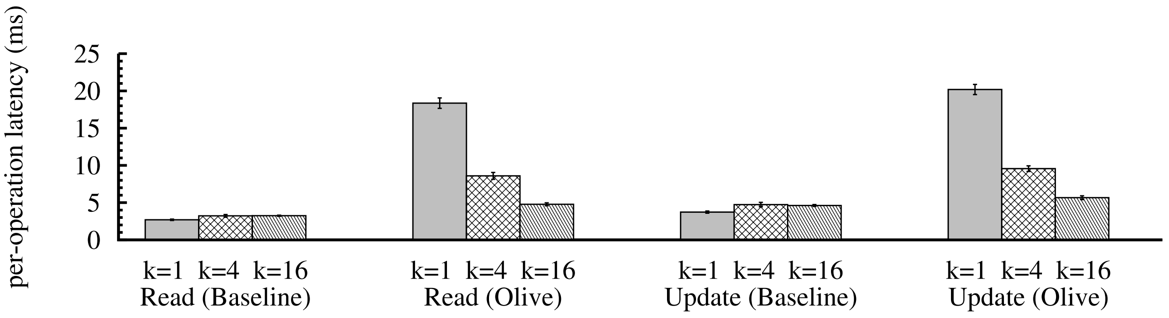


FIGURE 10—Per-operation latency when executing a sequence of *k* storage operations both normally and within an Olive intent. As the number of operations per intent increases, the per-intent costs (e.g., registering an intent, storing the final result, etc.) are amortized, and the per-operation latencies approach those of operations directly on the raw storage interface.

5.1 Implementation

We implement Olive as a client library in approximately 2,000 lines of C# code, including all features described in §3. As discussed there, although we have built an intent collector capable of coexisting with other instances of itself, our implementation does not support partitioning work among such instances for greater efficiency.

To allow Olive to work on multiple underlying storage systems, we implement it atop an abstract C# interface that exposes the storage model described in §2.1. We then build concrete C# classes that call cloud storage systems' APIs to implement the abstract interface. We implement two such concrete mappings. The first maps to Azure Table storage; it is only 38 lines of code because our abstract storage interface maps one-to-one to its API. The second maps to Amazon DynamoDB; it is 107 lines of code. DynamoDB provides atomicity at the granularity of individual objects [4], so our concrete class only allows `AtomicBatchUpdate` for object scopes.

5.2 Setup and method

We experiment on Olive with Microsoft Azure Table service as the cloud storage. For computation, we use a G3 VM instance (8-core Intel Xeon E5 v3 family with 112 GB RAM) running Windows Server 2012 R2 in the same availability zone as the storage service.

The principal goal of our evaluation is to understand the costs of robustness due to Olive's mechanisms relative to alternative mechanisms. To do so, we compare the performance of Olive-based artifacts with baselines providing similar fault-tolerance guarantees. For these comparisons, our performance metric is the latency of storage operations. In each experiment, we report the mean of at least 1,000 measurements along with the 95% confidence interval for that mean. For these end-to-end experiments, we use YCSB [25] to generate workloads.

5.3 Cost of Olive's exactly-once semantics

To understand the costs of Olive's logging for ensuring exactly-once semantics, we experiment with a series of microbenchmarks. We write two intents, one of which issues a single `Read` on an object and the other of which issues a single `Update`. Each object consists of a random 64-byte key and a random 1-KB value. Our baseline for this is a snippet of code that issues the same operations but without using Olive's intent-execution machinery. We run these intents and the associated baselines 1,000 times, and measure the latency of the aforementioned operations. Figure 9 depicts our results.

As expected, Olive pays significant latency overhead compared to a baseline that does not ensure exactly-once semantics. The reason is that Olive has to register its intent by writing to the `intents` table, then insert DAAL entries. Furthermore, our implementation writes an entry to another `results` table when an intent execution is complete. The last operation is not crucial to Olive, but stores a succinct summary of the intent execution including the final return value of the intent. Our implementation does this so that other clients can quickly learn the final return value of an intent by simply doing a lookup on this table.

Amortizing setup costs. Much of this overhead (registering an intent, saving the final results, etc.) is per-intent cost. Thus, to understand the costs of Olive's intent execution in a comprehensive manner, we run another set of experiments in which we vary the number of operations *k* per intent, setting $k=1,4$, and 16. Figure 10 depicts our results. As expected, the aforementioned per-intent costs amortize over multiple operations, and the per-operation cost of a storage operation in an intent is comparable to that of directly executing the operation without Olive.

Varying object sizes. We experiment with Olive and the baseline under varying value sizes (16 bytes, 128 bytes, and 1 KB) and with varying *k*. We find that neither Olive's costs nor the baseline's costs grow with value size, so we do not depict these results.

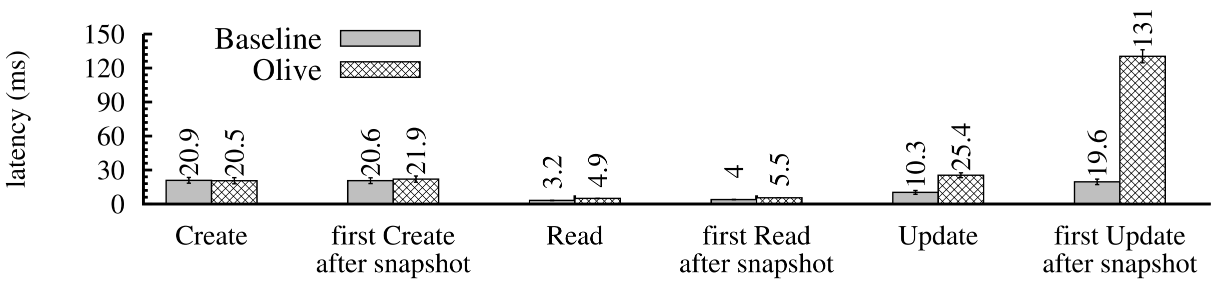


FIGURE 11—Latency of `Create`, `Read`, and `Update` operations using two snapshotting services: (i) a baseline that uses a cloud database's native support for creating snapshots, and (ii) the Olive-based artifact. Under Olive, the latency of an `Update` immediately after taking a snapshot takes 5× as long as a normal `Update`; the baseline incurs only 2× higher latency due to native support for snapshots. The Olive-based artifact is competitive with the baseline for non-`Update` operations.

5.4 End-to-end performance: snapshots

To understand the performance of an Olive-based artifact, we experiment with the snapshotting service we built. Two reasonable alternative approaches for building this artifact in a cloud environment are: (i) create a snapshotting service atop a cloud storage system similar to the Olive-based artifact, but, instead of using DAAL, put all the application data as well as snapshots of that data in the same atomicity scope; and (ii) employ a cloud storage system that natively supports creating snapshots, e.g., Azure SQL or Amazon Aurora.

Alternative (i) can exploit `AtomicBatchUpdate` to create snapshots of application data, without having to incur difficulties we discussed earlier (§2.2). Unfortunately, for most cloud applications, this is truly not an option. It severely limits throughput and scalability, because each atomicity scope supports only a few thousand requests per second. It also limits capacity, because each atomicity scope supports only a certain amount of data. Furthermore, in some cloud storage systems (e.g., Amazon DynamoDB), the atomicity scope is a single object, thereby rendering this option infeasible. We thus use alternative (ii) for our baseline; specifically, we use Azure SQL. Of course, this baseline supports far more features than our artifact, but we find that it is the closest alternative with similar functionality in a cloud environment.

In our measurements, a client process first preloads a table created by YCSB's benchmarking tool; the table contains 1,000 objects, each having ten attributes with 100-byte values. The client process then runs a series of experiments, in which it uses YCSB's core workloads a–d to generate a stream of 1,000 requests with varying mixtures of `Read`, `Create`, and `Update` operations. We also run another set of experiments, in which the client process creates a snapshot between preloading the table with data and running the workloads. This causes each `Update` operation in the experiments to perform copy-on-write. Figure 11 summarizes our results.

For operations other than `Update`, the Olive-based artifact's performance is competitive with the baseline. The largest difference is that, under Olive, the first `Update` immediately after taking a snapshot incurs 5× higher latency than a normal `Update`; for the baseline, it is only 2× higher. The primary reason is that, while the baseline requires only a single round trip to the database server, Olive incurs additional round trips and extra logging to ensure the exactly-once semantics. Furthermore, the database service likely uses complex machinery to implement the snapshotting feature efficiently.

Finally, because the Olive-based snapshotting service uses a NoSQL cloud storage system instead of the baseline's SQL database service, it likely incurs much lower monetary cost. Unfortunately, the pricing models of these cloud services are complex and hard to compare, so we now provide only a rough comparison.

Azure's table service charges on two axes: amount of data stored and number of operations performed. For example, in the West-US data center, it costs \$0.045–0.12 per GB of data per month (depending on amount of data stored and the desired geo-replication level), and \$0.036 per million cloud storage operations [2].

On the other hand, Azure SQL charges for desired throughput, measured in database transaction units (DTUs) [6], and amount of data stored. A DTU is much more complex than the number of cloud storage operations because it accounts for the number of disk operations and the amount of processing consumed by a SQL query. As an example, for 5 DTUs and 2 GB of data, the charge is \$5/month. This increases to \$465/month with 125 DTUs and 500 GB of data.

A rough comparison using these figures suggests that the cost of a SQL database is at least an order of magnitude higher than Azure's table store. As a result, the Olive-based artifact reaps lower monetary costs from its underlying store while providing a snapshotting feature with comparable performance.

5.5 End-to-end performance: live re-partitioning

To understand the benefits of the flexible isolation properties of Olive's locks with intents, we evaluate two Olive-based implementations of the live re-partitioning service (§4.2): one using the Olive-based transaction library (§4.4) and the other using intents directly for fine-granularity isolation. Such a comparison will also help demonstrate the flexibility of locks with intent: developers can build their service atop our transaction library first for simplicity and then later optimize it by writing that service with intents directly for performance.

We run experiments in which a client process first preloads a table with 1,000 objects and then issues a stream of `Create`, `Read`, and `Update` operations generated via YCSB's core workloads a–d. Figure 12 depicts the performance of the intent-based artifact and compares it with a transaction-based implementation. We find that in all cases the intent-based artifact performs better than, or as well as, the transaction-based one.

A notable scenario where Olive's locks with intent enable us to optimize the re-partitioning service is in the implementation of its `Update` operation. In the intent-based artifact, this operation does not need to lock any objects, but the transaction library cannot avoid locking. In particular, the intent-based artifact implements this by exploiting the `UpdateIfUnchanged` API supported by the underlying cloud storage system. As a result, the latency goes down by roughly 6×. Similarly, for `Create` and the data-migration routine, the intent-based artifact locks fewer objects, enabling it to achieve better performance than the transaction-based one.

Finally, because the migration routine in the re-partitioning service locks and migrates one object at a time, if a normal table operation (e.g., `Read`) observes that an object is locked, it has to block for the duration of data migration. (Figure 12 does not depict this case.)

5.6 Storage overheads

Up to this point, we have measured the latency overhead Olive incurs to ensure exactly-once semantics. We now evaluate Olive's storage overhead. We do this by running a microbenchmark with our re-partitioning service. In particular, we use the data-migration routine depicted in Figure 6. For the experiment, we preload data into a table and use YCSB's core workload a, which inserts 1,000 objects each having ten attributes with 100-byte values. We then run Olive's migration routine to move those objects to a new table. We measure the total size of all tables before and after the migration.

The application data inserted by the workload is roughly 1 MB. Before the migration runs, we find that the total size of the tables is 2.6 MB. The 2.6× overhead comes from the internal use of an intent in the table re-partitioning service's `Create` operation. This intent

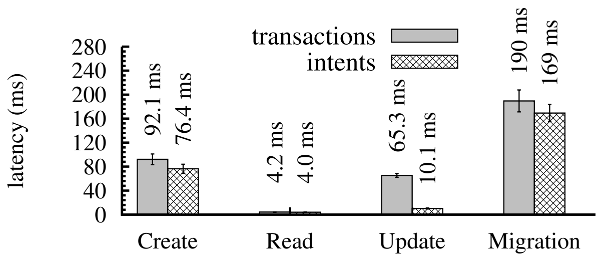


FIGURE 12—Comparison of the performance of table operations in two implementations of the re-partitioning service (§4.2), one using Olive's transaction library and another using Olive's intents directly. The intent-based artifact achieves better performance because we carefully optimize the set of objects that get locked, whereas the implementation based on Olive's transaction library (naively) locks all the objects affected by a table operation. (See text for details.)

serializes the object to be inserted and stores it in the `intents` table (as the initial state of the intent). As a result, it doubles the size of data in tables. The remaining overhead is due to DAAL entries and invisible attributes (e.g., `locked`). After the migration procedure completes, the total size of the data in the tables increases to 5.5 MB, which again is due to the use of intents. Fortunately, most of this overhead is ephemeral: after garbage collection and deletion of objects in the old table, the storage overhead is less than 8% of the application-data size.

5.7 Summary

While Olive incurs unavoidable latency and storage overheads to ensure exactly-once semantics (§5.3), our experience suggests that Olive's strong semantics make it easy to quickly build cloud services that are correct and fault-tolerant (§4). Furthermore, our end-to-end experiments show that Olive-based artifacts have performance comparable to baselines that provide similar fault-tolerance guarantees. Finally, even when a feature is offered by some cloud storage system, building the feature with Olive can save significant money by enabling the use of a less expensive cloud storage system.

6 Discussion

Comparison to transactions. Transactions are arguably simpler to use than intents, but offer less flexibility. Intents can support both strong consistency and eventual consistency, can avoid full isolation when not required, and can support exactly-once semantics which are often not provided by transactions. Indeed, as shown in §4.4, intents are general enough to support transactions, so developers who want a simple experience can always use the transaction library Olive provides.

Liveness of intents. The liveness of Olive's locks with intent hinges on the liveness of intents. Just as with code, developers must ensure that an intent does not contain bugs leading to infinite loops, crashes, or deadlocks. A more subtle concern is that Olive may amplify such bugs. For example, testing often misses bugs occurring in rare scenarios. So, Olive may trigger latent bugs by exercising the rare scenario in which an intent is executed not by its owner but by another client or the intent collector.

It is possible to automatically recover from deadlock bugs in intents. After all, Olive's locks with intent can be used to encode deadlock recovery logic inside an intent, e.g. by undoing the effects of the intent to release a lock. Olive's semantics ensure that such recovery logic inside an intent is executed exactly once. As an example, our OCC-based transaction library (§4.4) includes such logic to abort a transaction if it detects read-write conflicts during the verification step.

Garbage collection of intent logs. Entries in the `intents` table, as well as DAAL entries, need to be garbage-collected. But, this must be done with care, because a recovering client uses the existence of an entry to decide whether to execute a step in an intent. A slow client that attempts to execute a step in an intent might mistakenly re-execute it because the corresponding entries have been garbage collected. One solution is to introduce the notion of epochs for intents, with a mandate that an intent created in epoch n must be completed by the end of epoch $n+1$ and is considered *outdated* in epoch $n+2$ and beyond. It this case, is safe to garbage-collect entries for outdated intents. The exact duration of an epoch can be application-specific (e.g., a day).

Security and privacy. Olive assumes that all clients sharing an `intents` table belong to the same application, and thus any client can complete any other client's intent. However, for some applications this is not possible due to security restrictions. For instance, a client serving user Alice may downgrade its capabilities, to ensure it cannot accidentally leak information to Alice about other users. So, that client may be unable to complete an intent running on behalf of another user Bob.

Differences in clients' permissions can also lead to privacy violations. For instance, a client running on Alice's behalf may write Alice's private data into the `intents` table. Then, another client running on Bob's behalf may read her data from that table and leak it to Bob.

For these reasons, Olive is suitable only for applications with clients in equivalent security domains. In future work, we plan to address this limitation, e.g., by propagating clients' security restrictions to the `intents` table.

Cloud support. We have designed Olive under the assumption that cloud providers are unwilling or unable to change their APIs. However, a cloud provider could choose to add locks with intent to its external API. After all, unlike richer primitives like transactions, locks with intent would not require significant changes to cloud-storage internals. By adding locks with intent natively, and having more efficient execution paths for performing and completing intents, the cloud provider might achieve better performance than Olive.

7 Related work

Olive is related to a host of techniques that provide a substrate for building fault-tolerant services. It is also related to work that makes reasoning about concurrency easier for programmers.

State machine replication [38, 49] is a classic technique for building fault-tolerant services. A cloud application can use replication directly for fault tolerance. Olive instead takes a different approach by leveraging the underlying cloud storage, which is already made fault tolerant by using replication internally. Olive's approach avoids consensus at the application layer and maintains reliable persistent states only in cloud storage.

There is also a long line of work on recovering computation from failures [22–24, 42, 43]. Compared with this work, nodes in Olive maintain and share state via cloud storage, which makes recovering from failed computation harder. Even with microreboots [22, 23], maintaining persistent state consistently despite failures is left to applications, which Olive addresses.

Write-ahead logging [46] is a well-known technique, widely used in database systems [9, 14, 18, 47], to provide atomicity and durability in the presence of failures. Olive's intent `executionLog` is logically a write-ahead redo log, but Olive uses DAAL to achieve exactly-once semantics and to cope with concurrent executions of the same intent. A related technique, journaling, is also widely used in file system implementations [44, 45, 50] to ensure data consistency despite inopportune machine crashes.

Transactions, another popular primitive, provide strong ACID properties. Transactions simplify concurrency control by providing strong isolation [11, 17, 40, 41] among concurrently executing transactions. Recognizing the performance overheads imposed by general-purpose transactions in a distributed system, Sinfonia [13] proposes a restricted form of transactions called minitransactions. Sagas [31], on the other hand, proposes to split long-lived transactions into smaller pieces to enhance concurrency. It relies on user-defined compensating transactions to recover the database to a consistent state if individual transactions fail. Like Sagas, Salt [53] allows developers to improve performance by gradually weakening the semantics of performance-critical transactions. In the last few years, several works [12, 16, 26, 27, 55] have built distributed storage systems with general-purpose transactional features, sometimes exploiting modern hardware such as RDMA [27], a cluster of flash devices [15, 16], and TrueTime [26]. Olive takes a different approach since distributed transactions on cloud storage would be expensive. The lock with intent in Olive has the benefits of a transactional primitive (automated failure handling, robustness to concurrency), but exposes a simple concurrency primitive that can be implemented without the full machinery and expense of transactions.

Leases [32], another popular distributed-system primitive, ensure exclusive access to a data object for a configurable amount of time. Chubby [20] and ZooKeeper [10] each implements a reliable lock service with lease-like expiration, enabling nodes in a distributed system to coordinate, usually at a coarse granularity. Like Olive's locks with intent, the failure of a lease owner will not block the entire system forever, as leases eventually expire. However, when a lease owner crashes, lease expiration does not automatically restore cloud storage to a consistent state, a key problem that locks with intent address.

Revocable locks [34] provide the abstraction of non-blocking locks in a shared-memory model on a single machine. To get non-blocking semantics for locks, a thread can revoke a lock from its current owner and direct that lock owner's thread to execute a predefined recovery code block. Olive's locks with intent are similar in spirit. However, Olive does not require that the user write and reason about recovery logic. Also, Olive goes beyond a single-machine context to solve issues arising from machine failure and asynchrony in a distributed system. Such a design is crucial in Olive's context given the distributed-systems setting where accurate failure detection and synchronization among clients is hard.

Exactly-once semantics and idempotence have been recognized as critical properties in various systems for simplifying application development and achieving stronger semantics [35]. Exactly-one semantics has been used as a correctness criterion for replicated services [30], for building three-tier Web services [29], and for distributed message delivery systems [36]. It has also been incorporated into database systems via queued transaction processing [19]. Recognizing the power of such semantics, Ramalingam and Vaswani [48] design a programming language monad that uses idempotence and exactly-once semantics to tolerate process failures and message loss in a distributed system. However, they neither consider concurrency control primitives in the presence of failures, nor use automatic failure detection and retry mechanisms, leading to different design decisions. For example, in Olive's locks with intent, we find it crucial to track all intents associated with a locked object via cloud storage and to let any client execute any intent in the system. We also achieve a useful liveness property via an intent collector, which is not covered in their work.

In more recent work, RIFL [39] implements a reusable module to enhance the semantics of a key-value storage system's interface from at-least-once to exactly-once. An application that builds atop such a storage service can handle server failures more easily. Olive's locks with intent provide similar exactly-once semantics, but in a stronger sense: the improved semantics are useful to tolerate failures in the application layer, and they are guaranteed for arbitrary snippets of code rather than only for RPCs.

Besides fault tolerance and concurrency control, there are many works that enhance other properties of cloud services, such as the following. Tombolo [54] proposes the use of cloud gateways to reduce the latency of cloud data accesses. CosTLO [52] reduces the latency variance of cloud storage services. SPANStore [51] helps developers manage the use of multiple cloud storage services, to reduce service cost while still meeting the latency, data consistency, and fault tolerance requirements.

8 Conclusion

Cloud applications atop distributed reliable cloud storage services represent a new model of building fault-tolerant distributed systems, where all coordination at the application layer goes through cloud storage, without the need to re-implement consensus protocols.

Devising the right programming abstraction in this model involves the art of balancing a set of attributes, such as simplicity, programmability, expressiveness, efficiency, and generality. Olive's lock with intent strikes such a delicate balance: its exactly-once semantics and mutual exclusion property are simple to understand and to use when reasoning about correctness; it is easy for developers to program with because it reuses common constructs such as locks, with an intent just as an arbitrary code snippet; it can be used to implement both weak eventual consistency and strong transactional consistency, allowing an efficient design without excessive constraints; it is generally applicable to a set of cloud storage services and popular distributed storage systems with the use of a common storage API. The result is a powerful new primitive that allows us to develop a set of useful advanced functionalities easily, correctly, and efficiently.

Acknowledgments

We thank Mahesh Balakrishnan, Albert Greenberg, Rama Kotla, Ashwin Murthy, and Doug Terry for creating the Azure Replicated Table (RTable) project, which inspired us to investigate foundational primitives for building reliable cloud applications. We thank John Erickson and Matt McCutchen for introducing us to the live table repartitioning problem described in §4.2. We thank Sebastian Angel, Natacha Crooks, Thomas Moscibroda, Lenin Sivalingam, and the anonymous reviewers for their comments and discussions, which substantially improved this work. We are particularly grateful to our shepherd Peter Druschel for his insightful suggestions and guidance.

References

[1] Amazon DynamoDB. `https://aws.amazon.com/dynamodb/`.

[2] Azure Storage Pricing. `https://azure.microsoft.com/en-us/pricing/details/storage/tables/`.

[3] Azure Table storage. `https://azure.microsoft.com/en-us/documentation/articles/storage-dotnet-how-to-use-tables/`.

[4] BatchWriteItem. `http://docs.aws.amazon.com/amazondynamodb/latest/APIReference/API_BatchWriteItem.html`.

[5] CQL for Cassandra/BATCH. `https://docs.datastax.com/en/cql/3.3/cql/cql_reference/batch_r.html`.

[6] Database Transaction Units (DTUs). `https://azure.microsoft.com/en-us/documentation/articles/sql-database-what-is-a-dtu/`.

[7] Entity group transactions. `https://msdn.microsoft.com/en-us/library/azure/dd894038.aspx`.

[8] Google cloud Bigtable. `https://cloud.google.com/bigtable/docs/`.

[9] PostgreSQL. `http://www.postgresql.org/`.

[10] Apache ZooKeeper. `https://zookeeper.apache.org/`, 2008.

[11] A. Adya, B. Liskov, and P. O. Neil. Generalized isolation level definitions. In *International Conference on Data Engineering (ICDE)*, pages 67–78, 2000.

[12] M. K. Aguilera, J. B. Leners, and M. Walfish. Yesquel: scalable SQL storage for web applications. In *ACM Symposium on Operating Systems Principles (SOSP)*, 2015.

[13] M. K. Aguilera, A. Merchant, M. Shah, A. Veitch, and C. Karamanolis. Sinfonia: A new paradigm for building scalable distributed systems. In *ACM Symposium on Operating Systems Principles (SOSP)*, pages 159–174, 2007.

[14] M. M. Astrahan, M. W. Blasgen, D. D. Chamberlin, K. P. Eswaran, J. N. Gray, P. P. Griffiths, W. F. King, R. A. Lorie, P. R. McJones, J. W. Mehl, et al. System R: relational approach to database management. *ACM Transactions on Database Systems (TODS)*, 1(2):97–137, 1976.

[15] M. Balakrishnan, D. Malkhi, V. Prabhakaran, T. Wobbler, M. Wei, and J. D. Davis. CORFU: A shared log design for flash clusters. In *USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, pages 1–14, 2012.

[16] M. Balakrishnan, D. Malkhi, T. Wobber, M. Wu, V. Prabhakaran, M. Wei, J. D. Davis, S. Rao, T. Zou, and A. Zuck. Tango: Distributed data structures over a shared log. In *ACM Symposium on Operating Systems Principles (SOSP)*, pages 325–340, 2013.

[17] H. Berenson, P. Bernstein, J. Gray, J. Melton, E. O'Neil, and P. O'Neil. A critique of ANSI SQL isolation levels. In *ACM SIGMOD*, pages 1–10, 1995.

[18] P. A. Bernstein and E. Newcomer. *Principles of transaction processing*. Morgan Kaufmann, 2009.

[19] P. A. Bernstein and E. Newcomer. *Principles of transaction processing*. Morgan Kaufmann, 2009.

[20] M. Burrows. The Chubby lock service for loosely-coupled distributed systems. In *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2006.

[21] 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 *ACM Symposium on Operating Systems Principles (SOSP)*, pages 143–157, 2011.

[22] G. Candea, J. Cutler, and A. Fox. Improving availability with recursive microreboots: A soft-state system case study. *Perform. Eval.*, 56(1-4):213–248, Mar. 2004.

[23] G. Candea, S. Kawamoto, Y. Fujiki, G. Friedman, and A. Fox. Microreboot — a technique for cheap recovery. In *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2004.

[24] K. M. Chandy and C. V. Ramamoorthy. Rollback and recovery strategies for computer programs. *IEEE Transactions on Computers*, 100(6):546–556, 1972.

[25] 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)*, pages 143–154, 2010.

[26] J. C. Corbett, J. Dean, M. Epstein, A. Fikes, C. Frost, J. 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 *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 251–264, 2012.

[27] A. Dragojević, D. Narayanan, E. B. Nightingale, M. Renzelmann, A. Shamis, A. Badam, and M. Castro. No compromises: Distributed transactions with consistency, availability, and performance. In *ACM Symposium on Operating Systems Principles (SOSP)*, pages 54–70, 2015.

[28] A. J. Elmore, S. Das, D. Agrawal, and A. El Abbadi. Zephyr: Live migration in shared nothing databases for elastic cloud platforms. In *ACM SIGMOD*, pages 301–312, 2011.

[29] S. Frølund and R. Guerraoui. Transactional exactly-once. Technical report, Hewlett-Packard Laboratories, 1999.

[30] S. Frølund and R. Guerraoui. X-ability: A theory of replication. In *ACM Symposium on Principles of Distributed Computing (PODC)*, 2000.

[31] H. Garcia-Molina and K. Salem. Sagas. In *Proceedings of ICMD*, pages 249–259, 1987.

[32] C. G. Gray and D. R. Cheriton. Leases: An efficient fault-tolerant mechanism for distributed file cache consistency. In *ACM Symposium on Operating Systems Principles (SOSP)*, 1989.

[33] J. Gray. The transaction concept: Virtues and limitations (invited paper). In *International Conference on Very Large Data Bases (VLDB)*, pages 144–154, 1981.

[34] T. Harris and K. Fraser. Revocable locks for non-blocking programming. In *ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP)*, pages 72–82, 2005.

[35] P. Helland. Idempotence is not a medical condition. *Communications of the ACM*, 55(5):56–65, 2012.

[36] Y. Huang and H. Garcia-Molina. Exactly-once semantics in a replicated messaging system. In *International Conference on Data Engineering (ICDE)*, pages 3–12, 2001.

[37] H.-T. Kung and J. T. Robinson. On optimistic methods for concurrency control. *ACM Transactions on Database Systems (TODS)*, 6(2):213–226, 1981.

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

[39] C. Lee, S. J. Park, A. Kejriwal, S. Matsushita, and J. Ousterhout. Implementing linearizability at large scale and low latency. In *ACM Symposium on Operating Systems Principles (SOSP)*, pages 71–86, 2015.

[40] B. Liskov. Distributed programming in Argus. *Communications of the ACM*, 31(3):300–312, 1988.

[41] B. Liskov, D. Curtis, P. Johnson, and R. Scheifer. Implementation of Argus. In *ACM Symposium on Operating Systems Principles (SOSP)*, 1987.

[42] D. E. Lowell, S. Chandra, and P. M. Chen. Exploring failure transparency and the limits of generic recovery. In *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 20–20, 2000.

[43] D. E. Lowell and P. M. Chen. Discount checking: Transparent, low-overhead recovery for general applications. Technical report, Technical Report CSE-TR-410-99, University of Michigan, 1998.

[44] C. Mason. Journaling with ReisersFS. *Linux Journal*, 2001(82es):3, 2001.

[45] A. Mathur, M. Cao, S. Bhattacharya, A. Dilger, A. Tomas, and L. Vivier. The new ext4 filesystem: current status and future plans. In *Proceedings of the Linux symposium*, volume 2, pages 21–33, 2007.

[46] C. Mohan, D. Haderle, B. Lindsay, H. Pirahesh, and P. Schwarz. ARIES: a transaction recovery method supporting fine-granularity locking and partial rollbacks using write-ahead logging. *ACM Transactions on Database Systems (TODS)*, 17(1):94–162, 1992.

[47] M. A. Olson, K. Bostic, and M. I. Seltzer. Berkeley DB. In *USENIX Annual Technical Conference (USENIX ATC)*, pages 183–191, 1999.

[48] G. Ramalingam and K. Vaswani. Fault tolerance via idempotence. In *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, pages 249–262, 2013.

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

[50] S. C. Tweedie. Journaling the Linux ext2fs filesystem. In *The Fourth Annual Linux Expo*, 1998.

[51] Z. Wu, M. Butkiewicz, D. Perkins, E. Katz-Bassett, and H. V. Madhyastha. SPANStore: Cost-effective geo-replicated storage spanning multiple cloud services. In *ACM Symposium on Operating Systems Principles (SOSP)*, 2013.

[52] Z. Wu, C. Yu, and H. V. Madhyastha. CosTLO: Cost-effective redundancy for lower latency variance on cloud storage services. In *USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2015.

[53] C. Xie, C. Su, M. Kapritsos, Y. Wang, N. Yaghmazadeh, L. Alvisi, and P. Mahajan. Salt: Combining acid and base in a distributed database. In *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 495–509, 2014.

[54] S. Yang, K. Srinivasan, K. Udayashankar, S. Krishnan, J. Feng, Y. Zhang, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. Tombolo: Performance enhancements for cloud storage gateways. In *IEEE Conference on Massive Data Storage (MSST)*, 2016.

[55] I. Zhang, N. K. Sharma, A. Szekeres, A. Krishnamurthy, and D. R. K. Ports. Building consistent transactions with inconsistent replication. In *ACM Symposium on Operating Systems Principles (SOSP)*, pages 263–278, 2015.

Consolidating Concurrency Control and Consensus for Commits under Conflicts

Shuai Mu[⋆], Lamont Nelson[⋆], Wyatt Lloyd[†], and Jinyang Li[⋆]
[⋆]*New York University,* [†]*University of Southern California*

Abstract

Conventional fault-tolerant distributed transactions layer a traditional concurrency control protocol on top of the Paxos consensus protocol. This approach provides scalability, availability, and strong consistency. When used for wide-area storage, however, this approach incurs cross-data-center coordination twice, in serial: once for concurrency control, and then once for consensus. In this paper, we make the key observation that the coordination required for concurrency control and consensus is highly similar. Specifically, each tries to ensure the serialization graph of transactions is acyclic. We exploit this insight in the design of Janus, a unified concurrency control and consensus protocol. Janus targets one-shot transactions written as stored procedures, a common, but restricted, class of transactions. Like MDCC [16] and TAPIR [51], Janus can commit unconflicted transactions in this class in one round-trip. Unlike MDCC and TAPIR, Janus avoids aborts due to contention: it commits conflicted transactions in this class in at most two round-trips as long as the network is well behaved and a majority of each server replica is alive.

We compare Janus with layered designs and TAPIR under a variety of workloads in this class. Our evaluation shows that Janus achieves ~5× the throughput of a layered system and 90% of the throughput of TAPIR under a contention-free microbenchmark. When the workloads become contended, Janus provides much lower latency and higher throughput (up to 6.8×) than the baselines.

1 Introduction

Scalable, available, and strongly consistent distributed transactions are typically achieved through layering a traditional concurrency control protocol on top of shards of a data store that are replicated by the Paxos [19, 33] consensus protocol. Sharding the data into many small subsets that can be stored and served by different servers provides scalability. Replicating each shard with consensus provides availability despite server failure. Coordinating transactions across the replicated shards with concurrency control provides strong consistency despite conflicting data accesses by different transactions. This approach is commonly used in practice. For example, Spanner [9] implements two-phase-locking (2PL) and two phase commit (2PC) in which both the data and locks are replicated by Paxos. As another example, Percolator [35] implements a variant of opportunistic concurrency control (OCC) with 2PC on top of BigTable [7] which relies on primary-backup replication and Paxos.

When used for wide-area storage, the layering approach incurs cross-data-center coordination twice, in serial: once by the concurrency control protocol to ensure transaction consistency (strict serializability [34, 45]), and another time by the consensus protocol to ensure replica consistency (linearizability [14]). Such double coordination is not necessary and can be eliminated by consolidating concurrency control and consensus into a unified protocol. MDCC [16] and TAPIR [51] showed how unified approaches can provide the read-commited and strict serializability isolation levels, respectively. Both protocols optimistically attempt to commit and replicate a transaction in one wide-area roundtrip. If there are conflicts among concurrent transactions, however, they abort and retry, which can significantly degrade performance under contention. This concern is more pronounced for wide area storage: as transactions take much longer to complete, the amount of contention rises accordingly.

This paper proposes the Janus protocol for building fault-tolerant distributed transactions. Like TAPIR and MDCC, Janus can commit and replicate transactions in one cross-data-center roundtrip when there is no contention. In the face of interference by concurrent conflicting transactions, Janus takes at most one additional cross-data-center roundtrip to commit.

The key insight of Janus is to realize that strict serializability for transaction consistency and linearizability for replication consistency can both be mapped to the same underlying abstraction. In particular, both require the execution history of transactions be equivalent to some linear total order. This equivalence can be checked by constructing a serialization graph based on the execution history. Each vertex in the graph corresponds to a transaction. Each edge represents a dependency between two transactions due to replication or conflicting data access. Specifically, if transactions T_1 and T_2 make conflicting access to some data item, then there exists an edge $T_1 \rightarrow T_2$ if a shard responsible for the conflicted data item executes *or replicates* T_1 before T_2. An equivalent linear total order for both strict serializability and linearizability can be found if there are no cycles in this graph.

Janus ensures an acyclic serialization graph in two ways. First, Janus captures a preliminary graph by tracking the dependencies of conflicting transactions as they *arrive* at the servers that replicate each shard. The coordinator of a transaction T then collects and propagates T's dependencies from sufficiently large quorums of the servers that replicate each shard. The actual execution of T is deferred until the coordinator is certain that all of T's participating shards have obtained all dependencies for T. Second, although the dependency graph based on the arrival orders of transactions may contain cycles, T's participating shards re-order transactions in cycles in the same deterministic order to ensure an acyclic serialization graph. This provides strict serializability because all servers execute conflicting transactions in the same order.

In the absence of contention, all participating shards can obtain all dependencies for T in a single cross-data-center roundtrip. In the presence of contention, two types conflicts can appear: one among different transactions and the other during replication of the same transaction. Conflicts between different transactions can still be handled in a single cross-data-center roundtrip. They manifest as cycles in the dependency graph and are handled by ensuring deterministic execution re-ordering. Conflicts that appear during replication of the same transaction, however, require a second cross-data-center roundtrip to reach consensus among different server replicas with regard to the set of dependencies for the transaction. Neither scenario requires Janus to abort a transaction. Thus, Janus always commits with at most two cross-data-center roundtrips in the face of contention as long as network and machine failures do not conspire to prevent a majority of each server replica from communicating.

Janus's basic protocol and its ability to always commit assumes a common class of transactional workloads: one-shot transactions written as stored procedures. Stored procedures are frequently used [12, 13, 15, 22, 31, 40, 43, 46] and send transaction logic to servers for execution as pieces instead of having clients execute the logic while simply reading and writing data to servers. One-shot transactions preclude the execution output of one piece being used as input for a piece that executes on a different server. One-shot transactions are sufficient for many workloads, including the popular and canonical TPC-C benchmark. While the design of Janus can be extended to support general transactions, doing so comes at the cost of additional inter-server messages and the potential for aborts.

This approach of dependency tracking and execution re-ordering has been separately applied for concurrency control [31] and consensus [20, 30]. The insight of Janus is that both concurrency control and consensus can use a common dependency graph to achieve consolidation without aborts.

We have implemented Janus in a transactional key-value storage system. To make apples-to-apples comparisons with existing protocols, we also implemented 2PL+MultiPaxos, OCC+MultiPaxos and TAPIR [51] in the same framework. We ran experiments on Amazon EC2 across multiple availability regions using microbenchmarks and the popular TPC-C benchmark [1]. Our evaluation shows that Janus achieves ~5× the throughput of layered systems and 90% of the throughput of TAPIR under a contention-free microbenchmark. When the workloads become contended due to skew or wide-area latency, Janus provides much lower latency and more throughput (up to 6.8×) than existing systems by eliminating aborts.

2 Overview

This section describes the system setup we target, reviews background and motivation, and then provides a high level overview of how Janus unifies concurrency control and consensus.

2.1 System Setup

We target the wide-area setup where data is sharded across servers and each data shard is replicated in several geographically separate data centers to ensure availability. We assume each data center maintains a full replica of data, which is a common setup [4, 26]. This assumption is not strictly needed by existing protocols [9, 51] or Janus. But, it simplifies the performance discussion in terms of the number of cross-data-center roundtrips needed to commit a transaction.

We adopt the execution model where transactions are represented as *stored procedures*, which is also commonly used [12, 13, 15, 22, 31, 40, 43, 46]. As data is sharded across machines, a transaction is made up of a series of stored procedures, referred to as *pieces*, each of which accesses data items belonging to a single shard. The execution of each piece is atomic with regard to other concurrent pieces through local locking. The stored-procedure execution model is crucial for Janus to achieve good performance under contention. As Janus must serialize the execution of conflicting pieces, stored procedures allows the execution to occur at the servers,

System	Commit latency in wide-area RTTs	How to resolve conflicts during replication	How to resolve conflicts during execution/commit
2PL+MultiPaxos [9]	2*	Leader assigns ordering	Locking†
OCC+MultiPaxos [35]	2*	Leader assigns ordering	Abort and retry
Replicated Commit [28]	2	Leader assigns ordering	Abort and retry
TAPIR [51]	1	Abort and retry commit	Abort and retry
MDCC [16]	1	Retry via leader	Abort
Janus	1	Reorder	Reorder

Table 1: Wide-area commit latency and conflict resolution strategies for Janus and existing systems. MDCC provides read commited isolation, all other protocols provide strict serializability.

which is much more efficient than if the execution was carried out by the clients over the network.

2.2 Background and Motivation

The standard approach to building fault-tolerant distributed transactions is to layer a concurrency control protocol on top of a consensus protocol used for replication. The layered design addresses three paramount challenges facing distributed transactional storage: consistency, scalability and availability.

- **Consistency** refers to the ability to preclude "bad" interleaving of concurrent conflicting operations. There are two kinds of conflicts: 1) transactions containing multiple pieces performing non-serializable data access at different data shards. 2) transactions replicating their writes to the same data shard in different orders at different replica servers. Concurrency control handles the first kind of conflict to achieve strict serializability [45]; consensus handles the second kind to achieve linearizability [14].
- **Scalability** refers to the ability to improve performance by adding more machines. Concurrency control protocols naturally achieve scalability because they operate on individual data items that can be partitioned across machines. By contrast, consensus protocols do not address scalability as their state-machine approach duplicates all operations across replica servers.
- **Availability** refers to the ability to survive and recover from machine and network failures including crashes and arbitrary message delays. Solving the availability challenge is at the heart of consensus protocols while traditional concurrency control does not address it.

Because concurrency control and consensus each solves only two out of the above three challenges, it is natural to layer one on top of the other to cover all bases. For example, Spanner [9] layers 2PL/2PC over Paxos. Percolator [35] layers a variant of OCC over BigTable which relies on primary-backup replication and Paxos.

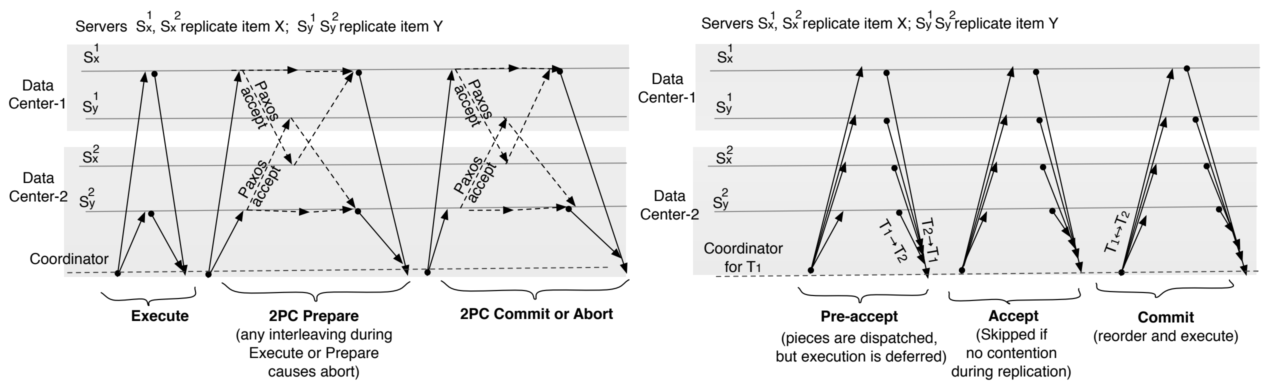
The layered design is simple conceptually, but does not provide the best performance. When one protocol is layered on top of the other, the coordination required for consistency occurs twice, serially; once at the concurrency control layer, and then once again at the consensus layer. This results in extra latency that is especially costly for wide area storage. This observation is not new and has been made by TAPIR and MDCC [16, 51], which point out that layering suffers from over-coordination.

A unification of concurrency control and consensus can reduce the overhead of coordination. Such unification is possible because the consistency models for concurrency control (strict serializability [34, 45]) and consensus (linearizability [14]) share great similarity. Specifically, both try to satisfy the ordering constraints among conflicting operations to ensure equivalence to a linear total ordering. Table 1 classifies the techniques used by existing systems to handle conflicts that occur during transaction execution/commit and during replication. As shown, optimistic schemes rely on abort and retry, while conservative schemes aim to avoid costly aborts. Among the conservative schemes, consensus protocols rely on the (Paxos) leader to assign the ordering of replication operations while transaction protocols use per-item locking.

In order to unify concurrency control and consensus, one must use a *common* coordination scheme to handle both types of conflicts in the same way. This design requirement precludes leader-based coordination since distributed transactions that rely on a single leader to assign ordering [41] do not scale well. MDCC [16] and TAPIR [51] achieve unification by relying on aborts and retries for both concurrency control and consensus. Such an optimistic approach is best suited for workloads with low contention. The goal of our work is to develop a unified approach that can work well under contention by avoiding aborts.

∗Additional roundtrips are required if the coordinator logs the commit status durably before returning to clients.

†2PL also aborts and retries due to false positives in distributed deadlock detection.

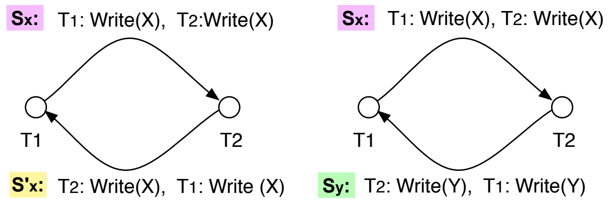

Servers S_x^1, S_x^2 replicate item X; S_y^1 S_y^2 replicate item Y
Data Center-1
Data Center-2
Coordinator
Paxos accept
Execute
2PC Prepare
(any interleaving during Execute or Prepare causes abort)
2PC Commit or Abort
Coordinator for T1
T1→T2
T2→T1
T1↔T2
Pre-accept
(pieces are dispatched, but execution is deferred)
Accept
(Skipped if no contention during replication)
Commit
(reorder and execute)

(a) OCC over Multi-Paxos. **(b) Janus**

Figure 1: Workflows for committing a transaction that increments x and y that are stored on different shards. The execute and 2PC-prepare messages in OCC over Multi-Paxos can be combined for stored procedures.

2.3 A Unified Approach to Concurrency Control and Consensus

Can we design unify concurrency control and consensus with a common coordination strategy that minimizes costly aborts? A key observation is that the ordering constraints desired by concurrency control and replication can both be captured by the same serialization graph abstraction. In the serialization graph, vertexes represent transactions and directed edges represent the dependencies among transactions due to the two types of conflicts. Both strict serializability and linearizability can be reduced to checking that this graph is free of cycles [20, 45].


Sx: T1: Write(X), T2:Write(X)
T1
T2
S'x: T2: Write(X), T1: Write (X)
Sx: T1: Write(X), T2: Write(X)
T1
T2
Sy: T2: Write(Y), T1: Write(Y)

(a) A cycle representing replication conflicts. **(b) A cycle representing transaction conflicts.**

Figure 2: Replication and transaction conflicts can be represented as cycles in a common serialization graph.

Janus attempts to explicitly capture a preliminary serialization graph by tracking the dependencies of conflicting operations without immediately executing them. Potential replication or transaction conflicts both manifest as cycles in the preliminary graph. The cycle in Figure 2a is due to conflicts during replication: T_1's write of data item X arrives before T_2's write on X at server S_x, resulting in $T_1 \rightarrow T_2$. The opposite arrival order happens at a different server replica S'_x, resulting in $T_1 \leftarrow T_2$. The cycle in Figure 2b is due to transaction conflicts: the server S_x receives T_1's write on item-x before T_2's write on item-x while server S_y receives the writes on item-y in the opposite order. In order to ensure a cycle-free serialization graph and abort-free execution, Janus deterministically re-orders transactions involved in a cycle before executing them.

To understand the advantage and challenges of unification, we contrast the workflow of Janus with that of OCC+MultiPaxos using a concrete example. The example transaction, T_1: x++; y++;, consists of two pieces, each is a stored procedure incrementing item-x or item-y.

Figure 1a shows how OCC+MultiPaxos executes and commits T_1. First, the coordinator of T_1 dispatches the two pieces to item-x and item-y's respective Paxos leaders, which happen to reside in different data centers. The leaders execute the pieces and buffer the writes locally. To commit T_1, the coordinator sends 2PC-prepare requests to Paxos leaders, which perform OCC validation and replicate the new values of x and y to others. Because the pieces are stored procedures, we can combine the dispatch and 2PC-prepare messages so that the coordinator can execute and commit T_1 in two cross-data-center roundtrips in the absence of conflicts. Suppose a concurrent transaction T_2 also tries to increment the same two counters. The dispatch messages (or 2PC-prepares) of T_1 and T_2 might be processed in different orders by different Paxos leaders, causing T_1 and/or T_2 to be aborted and retried. On the other hand, because Paxos leaders impose an ordering on replication, all replicas for item-x (or item-y) process the writes of T_1 or T_2 consistently, but at the cost of additional wide-area roundtrip.

Figure 1b shows the workflow of Janus. To commit and execute T_1, the coordinator moves through three phases: *PreAccept*, *Accept* and *Commit*, of which the *Accept* phase may be skipped. In *PreAccept*, the coordinator dispatches T_1's pieces to their corresponding replica servers. Unlike OCC+MultiPaxos, the server does not execute the pieces immediately but simply tracks their arrival orders in its local dependency graph. Servers reply to the coordinator with T_1's dependency information. The coordinator can skip the *Accept* phase if enough replica servers reply with identical dependencies, which happens when there is no contention among server replicas. Otherwise, the coordinator engages in another round of communication with servers to ensure that they obtain identical dependencies for T_1. In the *Commit* phase, each server executes transactions according to their dependencies. If there is a dependency cycle involving T_1, the server adheres to a deterministic order in executing transactions in the cycle. As soon as the coordinator hears back from the nearest server replica which typically resides in the local data center, it can return results to the user.

As shown in Figure 1b, Janus attempts to coordinate both transaction commit and replication in one shot with the *PreAccept* phase. In the absence of contention, Janus completes a transaction in one cross-data-center roundtrip, which is incurred by the *PreAccept* phase. (The commit phase incurs only local data-center latency.) In the presence of contention, Janus incurs two wide-area roundtrips from the *PreAccept* and *Accept* phases. These two wide-area roundtrips are sufficient for Janus to be able to resolve replication and transaction commit conflicts. The resolution is achieved by tracking the dependencies of transactions and then re-ordering the execution of pieces to avoid inconsistent interleavings. For example in Figure 1b, due to contention between T_1 and T_2, the coordinator propagates $T_1 \leftrightarrow T_2$ to all server replicas during commit. All servers detect the cycle $T_1 \leftrightarrow T_2$ and deterministically choose to execute T_1 before T_2.

Algorithm 1: Coordinator::DoTxn(T=[$\alpha_1, ..., \alpha_N$])

```
1  T's metadata is a globally unique identifier, a
     partition list, and an abandon flag.
2  PreAccept Phase:
3  send PreAccept($T$, $ballot$) to all participating
     servers, $ballot$ default as 0
4  if $\forall$ piece $\alpha_i \in \alpha_1 ... \alpha_N$, $\alpha_i$ has a fast quorum of
     PreAcceptOKs with the identical dependency $dep_i$
     then
5      $dep$ = Union($dep_1, dep_2, ..., dep_N$)
6      goto commit phase
7  else if $\forall$ $\alpha_i \in \alpha_1 ... \alpha_N$, $\alpha_i$ has a majority quorum of
     PreAcceptOKs then
8      foreach $\alpha_i \in \alpha_1 ... \alpha_N$ do
9          $dep_i$ ←Union dependencies returned by the
             majority quorum for piece $\alpha_i$.
10     $dep$ = Union($dep_1, dep_2, ..., dep_N$)
11     goto accept phase
12 else
13     return FailureRecovery(T)
14 Accept Phase:
15 foreach $\alpha_i$ in $\alpha_1 ... \alpha_N$ in parallel do
16     send Accept($T$, $dep$, $ballot$) to $\alpha_i$'s participating
         servers.
17 if $\forall$ $\alpha_i$ has a majority quorum of Accept-OKs then
18     goto commit phase
19 else
20     return FailureRecovery(T)
21 Commit Phase:
22 send Commit($T$, $dep$) to all participating servers
23 return to client after receiving execution results.
```

3 Design

This section describes the design for the basic Janus protocol. Janus is designed to handle a common class of transactions with two constraints: 1) the transactions do not have any user-level aborts. 2) the transactions do not contain any piece whose input is dependent on the execution of another piece, i.e., they are one-shot transactions [15]. Both the example in Section 2 and the popular TPC-C workload belong to this class of transactions. We discuss how to extend Janus to handle general transactions and the limitations of the extension in Section 4. Additional optimizations and a full TLA+ specification are included in a technical report [32].

3.1 Basic Protocol

The Janus system has three roles: clients, coordinators, and servers. A client sends a transaction request as a set of stored procedures to a nearest coordinator. The coordinator is responsible for communicating with servers storing the desired data shard to commit the transaction and then proxy the result to the client. Coordinators have no global nor persistent state and they do not communicate with each other. In our evaluation setup, coordinators are co-located with clients.

We require that the shards information of a transaction T is known before running. Suppose a transaction T involves n pieces, α_1,...,α_n. Let r be the number of servers replicating each shard. Thus, there are r servers processing each piece α_i. We refer to them as α_i's participating servers. We refer to the set of $r * n$ servers for all pieces as the transaction T's participating servers, or the servers that T involves.

There is a fast path and standard path of execution in Janus. When there is no contention, the coordinator takes the fast path which consists of two phases: *pre-accept* and *commit*. When there is contention, the coordinator takes

Algorithm 2: Server S::PreAccept(T, *ballot*)

```
24 if $highest\_ballot_S[T] < ballot$ then
25     return PreAccept-NotOK
26 $highest\_ballot_S[T] \leftarrow ballot$
27 //add $T$ to $\mathcal{G}_S$ if not already
28 AddNewTx($\mathcal{G}_S$, $T$)
29 $\mathcal{G}_S[T].status \leftarrow$ pre-accepted#$ballot$
30 $dep \leftarrow \mathcal{G}_S[T].dep$
31 return PreAccept-OK, $dep$
```

Algorithm 3: Server S::Accept(T, *dep*, *ballot*)

```
32 if $\mathcal{G}_S[T].status \geq$ committing or
33    $highest\_ballot_S[T] > ballot$ then
34     return Accept-NotOK, $highest\_ballot_S[T]$
35 $highest\_ballot_S[T'] \leftarrow ballot$
36 $\mathcal{G}_S[T'].dep \leftarrow dep$
37 $\mathcal{G}_S[T'].status \leftarrow$ accepted#$ballot$
38 return Accept-OK
```

Algorithm 4: Server S::Commit(T, *dep*)

```
39 $\mathcal{G}_S[T].dep \leftarrow dep$
40 $\mathcal{G}_S[T].status \leftarrow$ committing
41 Wait & Inquire Phase:
42 repeat
43     choose $T' \rightsquigarrow T$: $\mathcal{G}_S[T'].status <$ committing
44     if $T'$ does not involve $S$ then
45         send Inquire($T'$) to a server that $T'$ involves
46     wait until $\mathcal{G}_S[T'].status \geq$ committing
47 until $\forall T' \rightsquigarrow T$ in $\mathcal{G}_S$ : $\mathcal{G}_S[T'].status \geq$ committing
48 Execute Phase:
49 repeat
50     choose $T' \in \mathcal{G}_S$: ReadyToProcess($T'$) {
51     $scc \leftarrow$ StronglyConnectedComponent($\mathcal{G}_S$, $T'$)
52     for each $T''$ in DeterministicSort($scc$) do
53         if $T''$ involves $S$ and not $T''.abandon$ then
54             $T''.result \leftarrow$ execute $T''$
55         $processed_S[T''] \leftarrow true$
56 until $processed_S[T]$ is $true$
57 reply CommitOK, T.abandon, T.result
```

Algorithm 5: Server S::ReadyToProcess(T)

```
58 if $processed_S[T]$ or $\mathcal{G}_S[T].status <$ committing then
59     return $false$
60 $scc \leftarrow$ StronglyConnectedComponent($\mathcal{G}_S$, $T$)
61 for each $T' \notin scc$ and $T' \rightsquigarrow T$ do
62     if $processed_S[T'] \neq true$ then
63         return $false$
64 return $true$
```

the standard path of three phases: *pre-accept*, *accept* and *commit*, as shown in Figure 1b.

Dependency graphs. At a high level, the goal of *pre-accept* and *accept* phase is to propagate the necessary dependency information among participating servers. To track dependencies, each server S maintains a local dependency graph, $\mathcal{G}_S$. This is a directed graph where each vertex represents a transaction and each edge represents the chronological order between two conflicting transactions. The incoming edges for a vertex is stored in its *deps* value. Additionally, each vertex contains the status of the transaction, which captures the stage of the protocol the transaction is currently in. The basic protocol requires three vertex status types with the ranked order pre-accepted < accepted < committing. Servers and coordinators exchange and merge these graphs to ensure that the required dependencies reach the relevant servers before they execute a transaction.

In Janus, the key invariant for correctness is that all participating servers must obtain the exact same set of dependencies for a transaction T before executing T. This is challenging due to the interference from other concurrent conflicting transactions and from the failure recovery mechanism.

Next, we describe the fast path of the protocol, which is taken by the coordinator in the absence of interference by concurrent transactions. Psuedocode for the coordinator is shown in Algorithm 1.

The PreAccept phase. The coordinator sends *PreAccept* messages both to replicate transaction T and to establish a preliminary ordering for T among its participating servers. As shown in Algorithm 2, upon receiving the pre-accept for T, server S inserts a new vertex for T if one does not already exist in the local dependency graph $\mathcal{G}_s$. The newly inserted vertex for T has the status pre-accepted. An edge $T' \rightarrow T$ is inserted if an existing vertex T' in the graph and T both intend to make conflicting access to a common data item at server S. The server then replies *PreAccept-OK* with T's direct dependencies $\mathcal{G}_S[T].dep$ in the graph. The coordinator waits to collect a sufficient number of pre-accept replies from each involved shard. There are several scenarios (Algorithm 1, lines 4-11). In order to take the fast path, the coordinator must receive a *fast quorum* of replies containing the *same* dependency list for each shard. A fast quorum $\mathcal{F}$ in Janus contains *all* r replica servers.

The fast quorum concept is due to Lamport who originally applied it in the FastPaxos consensus protocol [21]. In consensus, fast quorum lets one skip the Paxos leader and directly send a proposed value to replica servers. In Fast Paxos, a fast quorum must contain at least three quarter replicas. By contrast, when applying the idea to unify consensus and concurrency control, the size of the fast quorum is increased to include all replica servers. Futhermore, the fast path of Janus has the additional requirement that the dependency list returned by the fast quorum must be the same. This ensures that the ancestor graph of transaction T on the standard path and during failure recovery always agrees with what is determined on the fast path at the time of T's execution.

The Commit phase. The coordinator aggregates the direct dependencies of every piece and sends it in a *Commit* message to all servers (Algorithm 1, lines 21-22). When server S receives the commit request for T (Algorithm 4), it replaces T's dependency list in its local graph $\mathcal{G}_S$ and upgrades the status of T to committing. In order to execute T, the server must ensure that all ancestors of T have the status committing. If server S participates in ancestor transaction T', it can simply wait for the commit message for T'. Otherwise, the server issues an Inquire request to a nearest server S' that participates in T' to request the direct dependencies $\mathcal{G}_{S'}[T'].dep$ after T' has become committing on S'.

In the absence of contention, the dependency graph at every server is acyclic. Thus, after all the ancestors of T become committing at server S, it can perform a topological sort on the graph and executes the transaction according to the sorted order. After executing transaction T, the server marks T as processed locally. After a server executes a transaction shard, it returns the result back to the coordinator. The coordinator can reply to the client as soon as it receives the necessary results from the nearest server replica, which usually resides in the local data center. Thus, on the fast path, a transaction can commit and execute with only one cross-data center round-trip, taken by the pre-accept phase.

3.2 Handling Contention Without Aborts

Contention manifests itself in two ways during the normal execution (Algorithm 1). As an example, consider two concurrent transactions T_1, T_2 of the form: `x++; y++`. First, the coordinator may fail to obtain a fast quorum of identical graphs for T_1's piece `x++` due to interference from T_2's pre-accept messages. Second, the dependency information accumulated by servers may contain cycles, e.g. with both $T_1 \rightsquigarrow T_2$ and $T_2 \rightsquigarrow T_1$ if the pre-accept messages of T_1 and T_2 arrive in different orders at different servers. Janus handles these two scenarios through the additional *accept* phase and deterministic re-ordering.

The Accept phase. If some shard does not return a fast quorum of identical dependency list during pre-accept, then consensus on the complete set of dependencies for T has not yet been reached. In particular, additional dependencies for T may be inserted later, or existing dependencies may be replaced (due to the failure recovery mechanism, Section 3.3).

The accept phase tries to reach consensus via the ballot accept/reject mechanism similar to the one used by Paxos. Because a higher ballot supersedes a lower one, the accepted status of a transaction includes a ballot number such that accepted#b < accepted#b', if ballot $b<b'$. The coordinator aggregates the dependency information collected from a majority quorum of *pre-accept*s and sends an *accept* message to all participating servers with ballot number 0 (Algorithm 1, lines 14-17). The server handles *accept*s as shown in Algorithm 3; It first checks whether T's status in its local dependency graph is committing and whether the highest ballot seen for T is greater than the one in the *accept* request. If so, the server rejects the accept request. Otherwise, the server replaces T's ancestor with the new one, updates its status, and its highest ballot seen for T. After,

the server replies *Accept-OK*. If the coordinator receives a majority quorum of *Accept-OK*s, it moves on to the commit phase. Otherwise, the accept phase has failed and the coordinator initiates the failure recovery mechanism (Section 3.3). In the absence of active failure recovery done by some other coordinator, the accept phase in Algorithm 1 always succeeds.

For a shard α_i, once the coordinator obtains a fast quorum of *PreAccept-OK*s with the same dependency list dep_i or a majority quorum of *Accept-OK*s accepting dependency list dep_i, then dep_i is the consensus dependency for piece α_i. When there are concurrent conflicting transactions, the dependencies for different shards $(dep_1, ..., dep_N)$ may not be identical. The coordinator aggregates them together and sends the resulting dependencies in the commit phase.

Deterministic execution ordering. In the general case with contention, the servers can observe cyclic dependencies among T and its ancestors after waiting for all T's ancestors to advance their status to committing. In this case, the server first computes all strongly connected components (SCCs) of $\mathcal{G}_S$ in T's ancestor graph. It then performs a topological sort across all SCCs and executes SCCs in sorted order. Each SCC contains one or more transactions. SCCs with multiple transactions are executed in an arbitrary, but deterministic order (e.g., sorted by transaction ids). Because all servers observe the same dependency graph and deterministically order transactions within a cycle, they execute conflicting transactions in the same order.

3.3 Handling Coordinator Failure

Algorithm 6: Coordinator C::FailureRecovery(T)

```
65 Prepare Phase:
66 $ballot \leftarrow$ highest ballot number seen + 1
67 send $Prepare(T, ballot)$ to $T$'s participating servers
68 if $\exists$ $Tx\text{-}Done, dep$ among replies then
69     //$T$'s direct dependencies $dep$ has been determined
70     goto commit phase
71 else if $\exists\ \alpha_i$ without a majority quorum of $Prepare\text{-}OK$s then
72     goto prepare phase
73 let $\mathcal{R}$ be the set of replies w/ the highest ballot
74 if $\exists dep' \in \mathcal{R}$: $dep'[T].status$ = accepted then
75     goto accept phase w/ $dep = dep'$
76 else if $\mathcal{R}$ contains at least $\mathcal{F} \cap \mathcal{M}$ identical dependencies $dep_i$ for each shard then
77     $dep$ = Union($dep_1$, $dep_2$, .. $dep_N$)
78     goto accept phase w/ $dep$
79 else if $\exists dep' \in \mathcal{R}$ : $dep'[T].status$ = pre-accepted then
80     goto preaccept phase, avoid fast path
81 else
82     $T.abandon = true$
83     goto accept phase w/ $dep = nil$
```

Algorithm 7: Server S::Prepare(T, $ballot$)

```
84 $dep \leftarrow \mathcal{G}_S[T].dep$
85 if $\mathcal{G}_S[T].status \geq$ commiting then
86     return $Tx\text{-}Done$, $dep$
87 else if $highest\_ballot_S[T] > ballot$ then
88     return $Prepare\text{-}NotOK$, $highest\_ballot_S[T]$
89 $highest\_ballot_S[T] \leftarrow ballot$
90 reply $T$, $Prepare\text{-}OK$, $\mathcal{G}_S[T].ballot$, $dep$
```

The coordinator may crash at anytime during protocol execution. Consequently, a participating server for transaction T will notice that T has failed to progress to the committing status for a threshold amount of time. This triggers the failure recovery process where some server takes up the role of the coordinator to try to commit T.

The recovery coordinator progresses through: *prepare*, *accept* and *commit* phases, as shown in Algorithm 6. Unlike in the normal execution (Algorithm 1), the recovery coordinator may repeat the *pre-accept*, *prepare* and *accept* phases many times using progressively higher ballot numbers, to cope with contention that arises when multiple servers try to recover the same transaction or when the original coordinator has been falsely suspected of failure.

In the *prepare* phase, the recovery coordinator picks a unique ballot number $b > 0$ and sends it to all of T's participating servers.[‡] Algorithm 7 shows how servers handle *prepare*s. If the status of T in the local graph is committing or beyond, the server replies *Tx-Done* with T's dependencies. Otherwise, the server checks whether it has seen a ballot number for T that is higher than that included in the prepare message. If so, it rejects. Otherwise, it replies *Prepare-OK* with T's direct dependencies. If the coordinator fails to receive a majority quorum of *Prepare-OK*s for some shard, it retries the prepare phase with a higher ballot after a back-off.

The coordinator continues if it receives a majority quorum ($\mathcal{M}$) of *Prepare-OK*s for each shard. Next, it must distinguish against several cases in order to decide whether to proceed from the pre-accept or the accept phase (lines 73-83). We point out a specific case in which the recovery coordinator has received a majority *Prepare-OKs* with the same dependencies for each shard and the status of T on the servers is pre-accepted. In this case, tranaction T could have succeeded on the fast path. Thus, the recovery coordinator merge the resulting dependencies and proceed to the accept phase. This case also illustrates why Janus must use a fast quorum $\mathcal{F}$ containing all server replicas. For any two conflicting transactions that both require recovery, a recovery coordinator is guaranteed to observe their dependency if $(\mathcal{F} \cap \mathcal{M}) \cap (\mathcal{F} \cap \mathcal{M}) \neq \emptyset$, where $\mathcal{M}$ is a majority quorum.

The pre-accept and accept phase used for recovery is the same as in the normal execution (Algorithm 1) except that both phases use the ballot number chosen in the prepare phase. If the coordinator receives a majority quorum of *Accept-OK* replies, it proceeds to the commit phase. Otherwise, it restarts in the prepare phase using a higher ballot number.

The recovery coordinator attempts to commit the transaction if possible, but may *abandon* it if the coordinator cannot recover the inputs for the transaction. This is possible when some servers pass on dependencies for one transaction to another and then fail simultaneously with the first transaction's coordinator. A recovery coordinator abandons a transaction T using the normal protocol except it sets the abandon flag for T during the accept phase and the commit phase. Abandoning a transaction this way ensures that servers reach consensus on abandoning T even if the the original coordinator is falsely suspected of failure. During transaction execution, if a server encounters a transaction T with the abandon flag set, it simply skips the execution of T.

[‡]Unique server ids can be appended as low order bits to ensure unique ballot numbers.

4 General Transactions

This section discusses how to extend Janus to handle dependent pieces and limited forms of user aborts. Although Janus avoids aborts completely for one-shot transactions, our proposed extensions incur aborts when transactions conflict.

Let a transaction's keys be the set of data items that the transaction needs to read or write. We classify general transactions into two categories: 1) transactions whose keys are known prior to execution, but the inputs of some pieces are dependent on the execution of other pieces. 2) transactions whose keys are not known beforehand, but are dependent on the execution of certain pieces.

Transactions with pre-determined keys. For any transaction T in this category, T's set of participating servers are known apriori. This allows the coordinator to go through the *pre-accept*, *accept* phases to fix T's position in the serialization graph while deferring piece execution. Suppose servers S_i, S_j are responsible for the data shards accessed by piece α_i and α_j ($i < j$) respectively and α_j's inputs depend on the outputs of α_i. In this case, we extend the basic protocol to allow servers to communicate with each other during transaction execution. Specifically, once server S_j is ready to execute α_j, it sends a message to server S_i to wait for the execution of α_i and fetch the corresponding outputs. We can use the same technique to handle transactions with a certain type of user-aborts. Specifically, the piece α_i containing user-aborts is not dependent on the execution of any other piece that performs writes. To execute any piece α_j of the transaction, server S_j communicates with server S_i to find out α_i's execution status. If α_i is aborted, then server S_j skips the execution of α_j.

Transactions with dependent keys. As an example transaction of this type, consider transaction T(x): $y \leftarrow$Read(x); Write(y, ...) The key to be accessed by T's second piece is dependent on the value read by the first piece. Thus, the coordinator cannot go through the usual phases to fix T's position in the serialization graph without execution. We support such a transaction by requiring users to transform it to transactions in the first category using a known technique [41]. In particular, any transaction with dependent keys can be re-written into a combination of several read-only transactions that determine the unknown keys and a conditional-write transaction that performs writes if the read values match the input keys. For example, the previous transaction T can be transformed into two transactions, T1(x): $y \leftarrow$ Read(x); return y; followed by T2(x,y): $y' \leftarrow$ Read(x); if $y' \neq y$ {abort;} else {Write(y, ...);} This technique is optimistic in that it turns concurrency conflicts into user-level aborts. However, as observed in [41], real-life OLTP workloads seldom involve key-dependencies on frequently updated data and thus would incur few user-level aborts due to conflicts.

5 Implementation

In order to evaluate Janus together with existing systems and enable an apples-to-apples comparison, we built a modular software framework that facilitates the construction and debugging of a variety of concurrency control and consensus protocols efficiently.

Our software framework consists of 36,161 lines of C++ code, excluding comments and blank lines. The framework includes a custom RPC library, an in-memory database, as well as test suits and several benchmarks. The RPC library uses asynchronous socket I/O (`epoll/kqueue`). It can passively batch RPC messages to the same machine by reading or writing multiple RPC messages with a single system call whenever possible. The framework also provides common library functions shared by most concurrency control and consensus protocols, such as a multi-phase coordinator, quorum computation, logical timestamps, epochs, database locks, etc. By providing this common functionality, the library simplifies the task of implementing a new concurrency control or consensus protocol. For example, a TAPIR implementation required only 1,209 lines of new code, and the Janus implementation required only 2,433 lines of new code. In addition to TAPIR and Janus we also implemented 2PL+MultiPaxos and OCC+MultiPaxos.

Our OCC implementation is the standard OCC with 2PC. It does not include the optimization to combine the execution phase with the 2PC-prepare phase. Our 2PL is different from conventional implementations in that it dispatches pieces in parallel in order to minimize execution latency in the wide area. This increases the chance for deadlocks significantly. We use the wound-wait protocol [37] (also used by Spanner [9]) to prevent deadlocks. With a contended workload and parallel dispatch, the wound-wait mechanism results in many false positives for deadlocks. In both OCC and 2PL, the coordinator does not make a unilateral decision to abort transactions. The MultiPaxos implementation inherits the common optimization of batching many messages to and from leaders from the passive batching mechanism in the RPC library.

Apart from the design described in Section 3, our implementation of Janus also includes the garbage collection mechanism to truncate the dependency graph as transactions finish. We have not implemented Janus's coordinator failure recovery mechanism nor the extensions to handle dependent pieces. Our implementations of 2PL/OCC+MultiPaxos and TAPIR also do not include failure recovery.

	Oregon	Ireland	Seoul
Oregon	0.9	140	122
Ireland		0.7	243
Seoul			1.6

Table 2: Ping latency between EC2 datacenters (ms).

6 Evaluation

Our evaluation aims to understand the strength and limitations of the consolidated approach of Janus. How does it compare against conventional layered systems? How does it compare with TAPIR's unified approach, which aborts under contention? The highlights include:

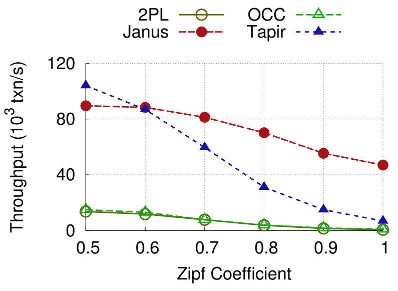
- In a single data center with no contention, Janus achieves 5× the throughput of 2PL/OCC+MultiPaxos, and 90% of TAPIR's throughput in microbenchmarks.
- All systems' throughput decrease as contention rises. However, as Janus avoids aborts, its performance under moderate or high contention is better than existing systems.
- Wide-area latency leads to higher contention. Thus, the relative performance difference between Janus and the other systems is higher in multi-data-center experiments than in single-data-center ones.

6.1 Experimental Setup

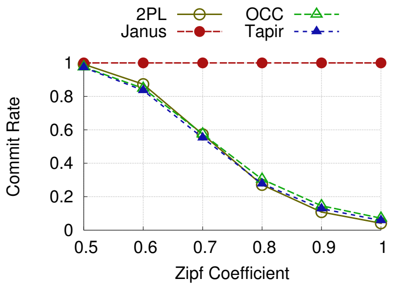
Testbed. We run all experiments on Amazon EC2 using m4.large instance types. Each node has 2 virtual CPU cores, 8GB RAM. For geo-replicated experiments, we use 3 EC2 availability regions, us-west-2 (Oregon), ap-northeast-2 (Seoul) and eu-west-1 (Ireland). The ping latencies among these data centers are shown in Table 2.

Experimental parameters. We adopt the configuration used by TAPIR [51] where a separate server replica group handles each data shard. Each microbenchmark uses 3 shards with a replication level of 3, resulting in a total of 9 server machines being used. In this setting, a majority quorum contains at least 2 servers and a fast quorum must contain all 3 servers. When running geo-replicated experiments, each server replica resides in a different data center. The TPC-C benchmark uses 6 shards with a replication level of 3, for a total of 18 processes running on 9 server machines.

We use closed-loop clients: each client thread issues one transaction at a time back-to-back. Aborted transactions are retried for up to 20 attempts. We vary the injected load by changing the number of clients. We run client processes on a separate set of EC2 instances than servers. In the geo-replicated setting, experiments of 2PL and OCC co-locate the clients in the datacenter containing the underlying MultiPaxos leader. Clients of Janus



(a) Cluster Throughput



(b) Commit Rates

Figure 3: Single datacenter performance with increasing contention as controlled by the Zipf coefficient.

and Tapir are spread evenly across the three datacenters. We use enough EC2 machines such that clients are not bottlenecked.

Other measurement details. Each of our experiment lasts 30 seconds, with the first 7.5 and last 7.5 seconds excluded from results to avoid start-up, cool-down, and potential client synchronization artifacts. For each set of experiments, we report the throughput, 90th percentile latency, and commit rate. The system throughput is calculated as the number of committed transactions divided by the measurement duration and is expressed in transactions per second (tps). We calculate the latency of a transaction as the amount of time taken for it to commit, including retries. A transaction that is given up after 20 retries is considered to have infinite latency. We calculate the commit rate as the total number of committed transactions divided by the total number of commit attempts (including retries).

Calibrating TAPIR's performance. Because we re-implemented the TAPIR protocol using a different code base, we calibrated our results by running the most basic experiment (one-key read-modify-write transactions), as presented in Figure 7 of TAPIR's technical report [49]. The experiments run within a single data center with only one shard. The keys are chosen uniformly randomly. Our TAPIR implementation (running on EC2) achieves a peak throughput of ~165.83K tps, which is much higher than the amount (~8K tps) reported for Google VMs [49]. We also did the experiment corresponding to Figure 13 of [51] and the results show a similar abort rate performance for TAPIR and OCC (TAPIR's abort rate is an order of magnitude lower than OCC with lower zipf coefficients). Therefore, we believe our TAPIR implementation is representative.

6.2 Microbenchmark

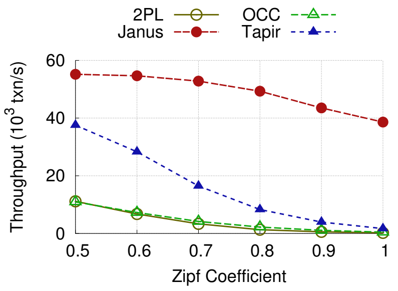
Workload. In the microbenchmark, each transaction performs 3 read-write access on different shards by incrementing 3 randomly chosen key-value pairs. We prepopulate each shard with 1 million key-value pairs before starting the experiments. We vary the amount of contention in the system by choosing keys according to a zipf distribution with a varying zipf coefficient. The larger the zipf coefficient, the higher the contention. This type of microbenchmark is commonly used in evaluations of transactional systems [9, 51].

Single data center (low contention). Figure 3 shows the single data center performance of different systems when the zipf coefficient increases from 0.5 to 1.0. Zipf coefficients of 0.0~0.5 are excluded because they have negligible contention and similar performance to zipf=0.5.

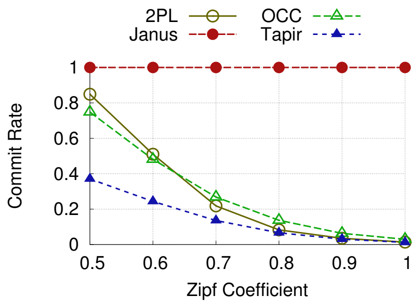
In these experiments, we use 900 clients to inject load. The number of clients is chosen to be on the "knee" of the latency-throughput curve of TAPIR for a specific zipf value (0.5). In other words, using more than 900 clients results in significantly increased latency with only small throughput improvements. With zipf coefficients of 0.5~0.6, the system experiences negligible amounts of contention. As Figure 3b shows, the commit rates across all systems are almost 1 with zipf coefficient 0.5.

Both TAPIR and Janus achieve much higher throughput than 2PL/OCC+MultiPaxos. As a Paxos leader needs to handle much more communication load (>3×) than non-leader server replicas, 2PL/OCC's performance is bottlenecked by Paxos leaders, which are only one-third of all 9 server machines.

Single data center (moderate to high contention). As the zipf value varies from 0.6 to 1.0, the amount of contention in the workload rises. As seen in Figure 3b, the commit rates of TAPIR, 2PL and OCC decrease quickly as the zipf value increases from 0.6 to 1.0. At first glance, it is surprising that 2PL's commit rate is no higher than OCC's. This is because our 2PL implementation dispatches pieces in parallel. This combined with the large amounts of false positives induced by deadlock detection, makes locking ineffective. By contrast, Janus does not incur any aborts and maintains a 100% commit rate. Increasing amounts of aborts result in significantly lower throughput for TAPIR, 2PL and OCC. Although Janus



(a) Cluster Throughput

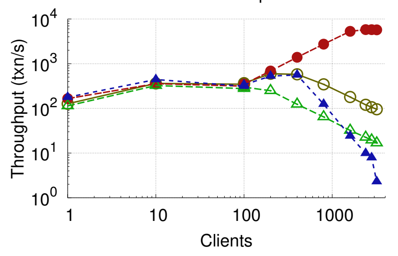


(b) Commit Rates

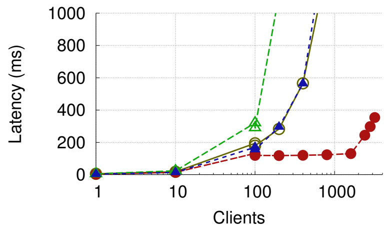
Figure 4: Geo-replicated performance with increasing contention as controlled by the Zipf coefficient.

does not have aborts, its throughput also drops because the size of the dependency graph that is being maintained and exchanged grows with the amount of the contention. Nevertheless, the overhead of graph computation is much less than the cost of aborts/retries, which allows Janus to outperform existing systems. At zipf=0.9, the throughput of Janus (55.41K tps) is 3.7× that of TAPIR (14.91K tps). The corresponding 90-percentile latency for Janus and TAPIR is 24.65ms and 28.90ms, respectively. The latter is higher due to repeated retries.

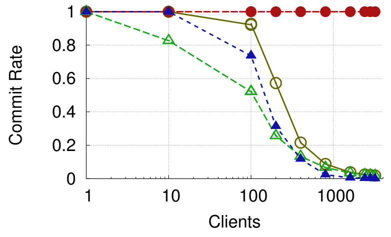
Multiple data centers (moderate to high contention). We move on to experiments that replicate data across multiple data centers. In this set of experiments, we use 10800 clients to inject load, compared to 900 for in the single data center setting. As the wide-area communication latency is more than an order of magnitude larger, we have to use many more concurrent requests in order to achieve high throughput. Thus, the amount of contention for a given zipf value is much higher in multi-datacenter experiments than that of single-data-center experiments. As we can see in Figure 4b, at zipf=0.5, the commit rate is only 0.37 for TAPIR. This causes the throughput of TAPIR to be lower than Janus at zipf=0.5 (Figure 4a). At larger zipf values, e.g., zipf=0.9, the throughput of Janus drops to 43.51K tps, compared to 3.95K tps for TAPIR, and ~1085 tps for 2PL and OCC. As Figure 4b shows, TAPIR's commit rate is slightly lower than that of 2PL/OCC. Interference during replication, apart from



(a) Cluster Throughput

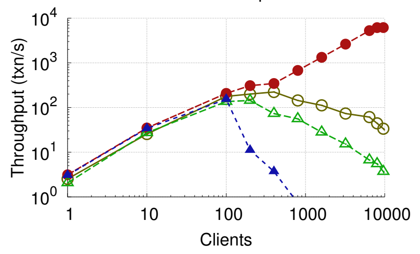


(b) 90% Latency

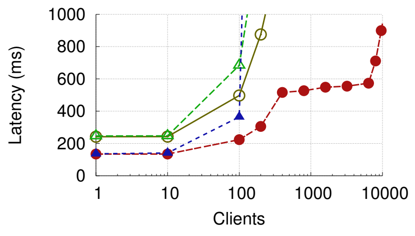


(c) Commit Rates

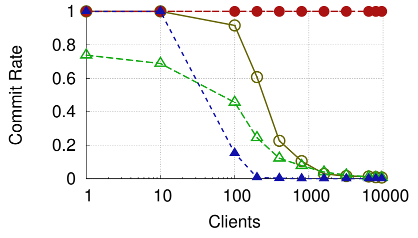
Figure 5: Performance in the single datacenter setting for the TPC-C benchmark with increasing load and contention as controlled by the number of clients per partition.



(a) Cluster Throughput



(b) 90% Latency



(c) Commit Rates

Figure 6: Performance in the geo-replicated setting for the TPC-C benchmark with increasing load and contention as controlled by the number of clients per partition.

aborts due to transaction conflicts, may also cause TAPIR to retry the commit.

6.3 TPC-C

	new-order	payment	order-status	delivery	stock-level
ratio	43.65%	44.05%	4.13%	3.99%	4.18%

Table 3: TPC-C mix ratio in a Janus trial.

Workload. As the most commonly accepted benchmark for testing OLTP systems, TPC-C has 9 tables and 92 columns in total. It has five transaction types, three of which are read-write transactions and two are read-only transactions. Table 3 shows a commit transaction ratio in one of our trials. In this test we use 6 shards each replicated in 3 data centers. The workload is sharded by warehouse; each shard contains 1 warehouse; each warehouse contains 10 districts, following the specification. Because each new-order transaction needs to do a read-modify-write operation on the next-order-id that is unique in a district, this workload exposes very high contention with increasing numbers of clients.

Single datacenter setting. Figure 5 shows the single data center performance of different systems when the number of clients is increased. As the number of clients increases, the throughput of Janus climbs until it saturates the servers' CPU, and then stabilizes at 5.78K tps. On the other hand, the throughput of other systems will first climb, and then drop due to high abort rates incurred as contention increases. TAPIR's the throughput peaks at 560 tps; The peak throughput for for 2PL/OCC is lower than 595/324 tps. Because TAPIR can abort and retry more quickly than OCC, its commit rate becomes lower than OCC as the contention increases. Because of the massive number of aborts, the 90-percentile latency for 2PL/OCC/TAPIR are more than several seconds when the number of clients increases; by contrast, the latency of Janus remains relatively low (<400ms). The latency of Janus increases with the number of clients due to as more outstanding transactions result in increased queueing.

Multi-data center setting. When replicating data across multiple data centers, more clients are needed to saturate the servers. As shown in Figure 6, the throughput of Janus (5.7K tps) is significantly higher than the other protocols. In this setup, the other systems show the same trend as in single data-center but the peak throughput becomes lower because the amount of contention increases dramatically with increased number of clients. TAPIR's commit rate is lower than OCC's when there are ≥ 100 clients because the wide-area setting leads to more aborts for its inconsistent replication.

Figure 6b shows the 90-percentile latency. When the number of clients is <= 100, the experiments use only a single client machine located in the Oregon data center. When the number of clients > 100, the clients are spread across all three data centers. As seen in Figure 6b, when the contention is low (<10 clients), the latency of TAPIR and Janus is low (less than 150ms) because both protocols can commit in one cross-data center roundtrip from Oregon. By contrast, the latency of 2PL/OCC is much more (230ms) as their commits require 2 cross data center roundtrips. In our experiments, all leaders of the underlying MultiPaxos happen to be co-located in the same data center as the clients. Thus, 2PL/OCC both require only one cross-data center roundtrip to complete the 2PC prepare phase and another roundtrip for the commit phase as our implementation of 2PL/OCC only returns to clients after the commit to reduce contention. As contention rises due to increased number of clients, the 90-percentile latency of 2PL/OCC/TAPIR increases quickly to tens of seconds. The latency of Janus also rises due to increased overhead in graph exchange and computation at higher levels of contention. Nevertheless, Janus achieves a decent 90-percentile latency (<900ms) as the number of clients increases to 10,000.

7 Related Work

We review related work in fault-tolerant replication, geo-replication, and concurrency control.

Fault-tolerant replication. The replicated state machine (RSM) approach [17, 38] enables multiple machines to maintain the same state through deterministic execution of the same commands and can be used to mask the failure of individual machines. Paxos [18] and viewstamped replication [24, 33] showed it was possible to implement RSMs that are always safe and remain live when f or fewer out of $2f + 1$ total machines fail. Paxos and viewstamped replication solve consensus for the series of commands the RSM executes. Fast Paxos [21] reduced the latency for consensus by having client send commands directly to replicas instead of a through a distinguished proposer. Generalized Paxos [20] builds on Fast Paxos by further enabling commands to commit out of order when they do not interfere, i.e., conflict. Janus uses this insight from Generalized Paxos to avoid specifying an order for non-conflicting transactions.

Mencius [29] showed how to reach consensus with low latency under low load and high throughput under high load in a geo-replicated setting by efficiently round-robining leader duties between replicas. Speculative Paxos [36] can achieve consensus in a single round trip by exploiting a co-designed datacenter network for consistent ordering. EPaxos [30] is the consensus protocol most related to our work. EPaxos builds on Mencius, Fast Paxos, and Generalized Paxos to achieve (near-)optimal commit latency in the wide-area, high throughput, and tolerance to slow nodes. EPaxos's use of dependency graphs to dynamically order commands based on their arrival order at different replicas inspired Janus's dependency tracking for replication.

Janus addresses a different problem than RSM because it provides fault tolerance and scalability to many shards. RSMs are designed to replicate a single shard and when used across shards they are still limited to the throughput of a single machine.

Geo-replication. Many recent systems have been designed for the geo-replicated setting with varying degrees of consistency guarantees and transaction support. Dynamo [11], PNUTS [8], and TAO [6] provide eventual consistency and avoid wide-area messages for most operations. COPS [26] and Eiger [27] provide causal consistency, read-only transactions, and Eiger provides write-only transactions while always avoiding wide-area messages. Walter [39] and Lynx [52] often avoid wide-area messages and provide general transactions with parallel snapshot isolation and serializability respectively. All of these systems will typically provide lower latency than Janus because they made a different choice in the fundamental trade off between latency and consistency [5, 23]. Janus is on the other side of that divide and provides strict serializability and general transactions.

Concurrency control. Fault tolerant, scalable systems typically layer a concurrency control protocol on top of a replication protocol. Sinfonia [3] piggybacks OCC into 2PC over primary-backup replication. Percolator [35] also uses OCC over primary-backup replication. Spanner [9] uses 2PL with wound wait over Multi-Paxos and inspired our 2PL experimental baseline.

CLOCC [2, 25] using fine-grained optimistic concurrency control using loosely synchronized clocks over viewstamped replication. Granola [10] is optimized for single shard transactions, but also includes a global transaction protocol that uses a custom 2PC over viewstamped replication. Calvin [42] uses a sequencing layer and 2PL over Paxos. Salt [47] is a concurrency control protocol for mixing acid and base transactions that inherits MySQL Cluster's chain replication [44]. Salt's successor, Callas, introduces modular concurrency control [48] that enables different types of concurrency control for different parts of a workload.

Replicated Commit [28] executes Paxos over 2PC instead of the typical 2PC over Paxos to reduce wide-area messages. Rococo [31] is a dependency graph based concurrency control protocol over Paxos that avoids aborts by reordering conflicting transactions. Rococo's protocol inspired our use of dependency tracking for concurrency control. All of these systems layer a concurrency control protocol over a separate replication protocol, which incurs coordination twice in serial.

MDCC [16] and TAPIR [49, 50, 51] are the most related systems and we have discussed them extensively throughout the paper. Their fast fault tolerant transaction commits under low contention inspired us to work on fast commits under all levels of contention, which is our biggest distinction from them.

8 Conclusion

We presented the design, implementation, and evaluation of Janus, a new protocol for fault tolerant distributed transactions that are one-shot and written as stored procedures. The key insight behind Janus is that the coordination required for concurrency control and consensus is highly similar. We exploit this insight by tracking conflicting arrival orders of both different transactions across shards and the same transaction within a shard using a single dependency graph. This enables Janus to commit in a single round trip when there is no contention. When there is contention Janus is able to commit by have all shards of a transaction reach consensus on its dependencies and then breaking cycles through deterministic reordering before execution. This enables Janus to provide higher throughput and lower latency than the state of the art when workloads have moderate to high skew, are geo-replicated, and are realistically complex.

Acknowledgments

The work is supported by NSF grants CNS-1514422 and CNS-1218117 as well as AFOSR grant FA9550-15-1-0302. Lamont Nelson is also supported by an NSF graduate fellowship. We thank Michael Walfish for helping us clarify the design of Janus and Frank Dabek for reading an early draft. We thank our shepherd, Dan Ports, and the anonymous reviewers of the OSDI program committee for their helpful comments.

References

[1] TPC-C Benchmark. http://www.tpc.org/tpcc/.

[2] A. Adya, R. Gruber, B. Liskov, and U. Maheshwari. Efficient optimistic concurrency control using loosely synchronized clocks. In *Proceedings of ACM International Conference on Management of Data (SIGMOD)*, 1995.

[3] M. K. Aguilera, A. Merchant, M. Shah, A. Veitch, and C. Karamanolis. Sinfonia: a new paradigm for building scalable distributed systems. In *Proceedings of ACM Symposium on Operating Systems Principles (SOSP)*, 2007.

[4] Amazon. Cross-Region Replication Using DynamoDB Streams. `http://docs.aws.amazon.com/amazondynamodb/latest/developerguide/Streams.CrossRegionRepl.html`, 2016.

[5] H. Attiya and J. L. Welch. Sequential consistency versus linearizability. *ACM Transactions on Computer Systems (TOCS)*, 12(2), 1994.

[6] 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 *Proceedings of USENIX Conference on Annual Technical Conference (ATC)*, 2013.

[7] F. Chang, J. Dean, S. Ghemawat, W. C. Hsieh, D. A. Wallach, M. Burrows, T. Chandra, A. Fikes, and R. E. Gruber. Bigtable: A distributed storage system for structured data. In *Proceedings of USENIX Symposium on Opearting Systems Design and Implementation (OSDI)*, 2006.

[8] 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. *Proceedings of International Conference on Very Large Data Bases (VLDB)*, 2008.

[9] 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 *Proceedings of USENIX Symposium on Opearting Systems Design and Implementation (OSDI)*, 2012.

[10] J. Cowling and B. Liskov. Granola: low-overhead distributed transaction coordination. In *Proceedings of USENIX Conference on Annual Technical Conference (ATC)*, 2012.

[11] G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, and W. Vogels. Dynamo: Amazon's highly available key-value store. In *Proceedings of ACM Symposium on Operating Systems Principles (SOSP)*, 2007.

[12] H. Garcia-Molina and K. Salem. Main memory database systems: An overview. *Knowledge and Data Engineering, IEEE Transactions on*, 4(6):509–516, 1992.

[13] H. Garcia-Molina, R. J. Lipton, and J. Valdes. A massive memory machine. *Computers, IEEE Transactions on*, 100 (5):391–399, 1984.

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

[15] R. Kallman, H. Kimura, J. Natkins, A. Pavlo, A. Rasin, S. Zdonik, E. P. Jones, S. Madden, M. Stonebraker,

Y. Zhang, et al. H-store: a high-performance, distributed main memory transaction processing system. In *Proceedings of International Conference on Very Large Data Bases (VLDB)*, 2008.

[16] T. Kraska, G. Pang, M. J. Franklin, S. Madden, and A. Fekete. MDCC: Multi-data center consistency. In *Proceedings of ACM European Conference on Computer Systems (EuroSys)*, 2013.

[17] L. Lamport. Time, clocks, and the ordering of events in a distributed system. *Commun. ACM*, 1978.

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

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

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

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

[22] K. Li and J. F. Naughton. Multiprocessor main memory transaction processing. In *Proceedings of the first international symposium on Databases in parallel and distributed systems*, 2000.

[23] R. J. Lipton and J. S. Sandberg. PRAM: A scalable shared memory. Technical Report TR-180-88, Princeton Univ., Dept. Comp. Sci., 1988.

[24] B. Liskov and J. Cowling. Viewstamped replication revisited. Technical report, MIT-CSAIL-TR-2012-021, MIT, 2012.

[25] B. Liskov, M. Castro, L. Shrira, and A. Adya. Providing persistent objects in distributed systems. In *ECOOP*, 1999.

[26] W. Lloyd, M. J. Freedman, M. Kaminsky, and D. G. Andersen. Don't settle for eventual: scalable causal consistency for wide-area storage with COPS. In *Proceedings of ACM Symposium on Operating Systems Principles (SOSP)*, 2011.

[27] W. Lloyd, M. J. Freedman, M. Kaminsky, and D. G. Andersen. Stronger semantics for low-latency geo-replicated storage. In *Proceedings of USENIX Conference on Networked Systems Design and Implementation (NSDI)*, 2013.

[28] H. Mahmoud, F. Nawab, A. Pucher, D. Agrawal, and A. El Abbadi. Low-latency multi-datacenter databases using replicated commit. In *Proceedings of International Conference on Very Large Data Bases (VLDB)*, 2013.

[29] Y. Mao, F. P. Junqueira, and K. Marzullo. Mencius: building efficient replicated state machines for wans. In *Proceedings of USENIX Symposium on Opearting Systems Design and Implementation (OSDI)*, 2008.

[30] I. Moraru, D. G. Andersen, and M. Kaminsky. There is more consensus in egalitarian parliaments. In *Proceedings of ACM Symposium on Operating Systems Principles (SOSP)*, 2013.

[31] S. Mu, Y. Cui, Y. Zhang, W. Lloyd, and J. Li. Extracting more concurrency from distributed transactions. In *Proceedings of USENIX Symposium on Opearting Systems Design and Implementation (OSDI)*, 2014.

[32] S. Mu, L. Nelson, , W. Lloyd, and J. Li. Consolidating concurrency control and consensus for commits under conflicts. Technical Report TR2016-983, New York University, Courant Institute of Mathematical Sciences, 2016.

[33] B. M. Oki and B. H. Liskov. Viewstamped replication: A new primary copy method to support highly-available distributed systems. In *Proceedings of ACM Symposium on Principles of Distributed Computing (PODC)*, 1988.

[34] C. H. Papadimitriou. The serializability of concurrent database updates. *Journal of the ACM*, 26(4), 1979.

[35] D. Peng and F. Dabek. Large-scale incremental processing using distributed transactions and notifications. In *Proceedings of USENIX Symposium on Opearting Systems Design and Implementation (OSDI)*, 2010.

[36] D. R. Ports, J. Li, V. Liu, N. K. Sharma, and A. Krishnamurthy. Designing distributed systems using approximate synchrony in data center networks. In *Proceedings of USENIX Conference on Networked Systems Design and Implementation (NSDI)*, 2015.

[37] D. J. Rosenkrantz, R. E. Stearns, and P. M. Lewis II. System level concurrency control for distributed database systems. *ACM Transactions on Database Systems (TODS)*, 3(2):178–198, 1978.

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

[39] Y. Sovran, R. Power, M. K. Aguilera, and J. Li. Transactional storage for geo-replicated systems. In *Proceedings of ACM Symposium on Operating Systems Principles (SOSP)*, 2011.

[40] 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 International Conference on Very Large Data Bases (VLDB)*, 2007.

[41] A. Thomson and D. J. Abadi. The case for determinism in database systems. In *Proceedings of International Conference on Very Large Data Bases (VLDB)*, 2010.

[42] A. Thomson, T. Diamond, S.-C. Weng, K. Ren, P. Shao, and D. J. Abadi. Calvin: fast distributed transactions for partitioned database systems. In *Proceedings of ACM International Conference on Management of Data (SIGMOD)*, 2012.

[43] S. Tu, W. Zheng, E. Kohler, B. Liskov, and S. Madden. Speedy transactions in multicore in-memory databases. In *Proceedings of ACM Symposium on Operating Systems Principles (SOSP)*, 2013.

[44] R. van Renesse and F. B. Schneider. Chain replication for supporting high throughput and availability. In *Proceedings of USENIX Symposium on Opearting Systems Design and Implementation (OSDI)*, 2004.

[45] G. Weikum and G. Vossen. *Transactional Information Systems: Theory, Algorithms, and the Practice of Concurrency Control and Recovery*. Morgan Kaufmann, 2001.

[46] A. Whitney, D. Shasha, and S. Apter. High volume transaction processing without concurrency control, two phase commit, sql or C++. In *Seventh International Workshop on High Performance Transaction Systems, Asilomar*, 1997.

[47] C. Xie, C. Su, M. Kapritsos, Y. Wang, N. Yaghmazadeh, L. Alvisi, and P. Mahajan. Salt: Combining ACID and BASE in a distributed batabase. In *Proceedings of USENIX Symposium on Opearting Systems Design and Implementation (OSDI)*, 2014.

[48] C. Xie, C. Su, C. Littley, L. Alvisi, M. Kapritsos, and Y. Wang. High-performance ACID via modular concurrency control. In *Proceedings of ACM Symposium on Operating Systems Principles (SOSP)*, 2015.

[49] I. Zhang, N. K. Sharma, A. Szekeres, A. Krishnamurthy, and D. R. Ports. Building consistent transactions with inconsistent replication. Technical report, Technical Report UW-CSE-2014-12-01, University of Washington, 2014.

[50] I. Zhang, N. K. Sharma, A. Szekeres, A. Krishnamurthy, and D. R. Ports. Building consistent transactions with inconsistent replication (extended version). Technical report, Technical Report UW-CSE-2014-12-01 v2, University of Washington, 2015.

[51] I. Zhang, N. K. Sharma, A. Szekeres, A. Krishnamurthy, and D. R. K. Ports. Building consistent transactions with inconsistent replication. In *Proceedings of ACM Symposium on Operating Systems Principles (SOSP)*, 2015.

[52] Y. Zhang, R. Power, S. Zhou, Y. Sovran, M. K. Aguilera, and J. Li. Transaction chains: achieving serializability with low latency in geo-distributed storage systems. In *Proceedings of ACM Symposium on Operating Systems Principles (SOSP)*, 2013.

Ryoan: A Distributed Sandbox for Untrusted Computation on Secret Data

Tyler Hunt, Zhiting Zhu, Yuanzhong Xu, Simon Peter, Emmett Witchel

The University of Texas at Austin

Abstract

Users of modern data-processing services such as tax preparation or genomic screening are forced to trust them with data that the users wish to keep secret. Ryoan protects secret data while it is processed by services that the data owner does not trust. Accomplishing this goal in a distributed setting is difficult because the user has no control over the service providers or the computational platform. Confining code to prevent it from leaking secrets is notoriously difficult, but Ryoan benefits from new hardware and a request-oriented data model.

Ryoan provides a distributed sandbox, leveraging hardware enclaves (e.g., Intel's software guard extensions (SGX) [15]) to protect sandbox instances from potentially malicious computing platforms. The protected sandbox instances confine untrusted data-processing modules to prevent leakage of the user's input data. Ryoan is designed for a request-oriented data model, where confined modules only process input once and do not persist state about the input. We present the design and prototype implementation of Ryoan and evaluate it on a series of challenging problems including email filtering, heath analysis, image processing and machine translation.

1. Introduction

Data-processing services are widely available on the Internet. Individual users can conveniently access them for tasks including image editing (Pixlr), tax preparation (TurboTax), data analytics (SAS OnDemand) and even personal health analysis (23andMe). However, user inputs to such services are often sensitive, such as tax documents and health data, which creates a dilemma for the user. In order to leverage the convenience and expertise of these services, she has to disclose sensitive data to them, potentially allowing them to disclose the data to further parties. If she wants to keep her data secret, she either has to give up using the services or hope that they can be trusted—that their service software will not leak data (intentionally or unintentionally), and that their administrators will not read the data while it resides on the server machines.

Companies providing data-processing services for users often wish to outsource part of the computation to third-party cloud services, a practice called "software as a service (SaaS)." For example, 23andMe may choose to use a general-purpose machine learning service hosted by Amazon. SaaS encourages the decomposition of problems into specialized pieces that can be assembled on behalf of a user, e.g., combining the health expertise of 23andMe with the machine learning expertise and robust cloud infrastructure of Amazon. However, 23andMe now finds itself a user of Amazon's machine learning service and faces its own dilemma—it must disclose proprietary correlations between health data and various diseases in order to use Amazon's machine learning service. In these scenarios, the owner of secret data has no control over the data-processing service.

We propose Ryoan[1], a distributed sandbox that allows users to keep their data secret in data-processing services, without trusting the software stack, developers, or administrators of these services. First, it provides a sandbox to confine individual data-processing modules and prevent them from leaking data; second, it uses trusted hardware to allow a remote user to verify the integrity of individual sandbox instances and protect their execution; third, the sandbox can be configured to allow confined code modules to communicate in controlled ways, enabling flexible delegation among mutually distrustful parties. Ryoan gives a user confidence that a service has protected her secrets.

A key enabling technology for Ryoan is hardware enclave-protected execution (e.g., Intel's software guard extensions (SGX) [15]), a new hardware primitive that uses trusted hardware to protect a user-level computation from potentially malicious privileged software. The processor hardware keeps unencrypted data on chip, but encrypts data when it moves into RAM. The hypervisor and operating system retain their ability to manage memory (e.g., move memory pages onto secondary storage), but privileged software sees only an encrypted version of the data that is protected from tampering by a cryptographic hash. Haven [21] and SCONE [19] are examples of systems that use enclaves to protect a user's computation from potentially malicious system software, including a library operating system to increase backward compatibility.

Ryoan faces issues beyond those faced by enclave-protected computation such as Haven [21]. Enclaves are intended to protect an application that is trusted by the user, which does not collude with the infrastructure, though it may unintentionally leak data via side channels. In Ryoan's model, neither the application nor the infrastructure is under the control of the user, and they may try to steal the user's secrets by colluding via *covert channels*—even if the application itself is isolated from

[1] Ryoan is a sandbox and its name is inspired by a famous dry landscape Zen garden that stimulates contemplation (Ryōan-ji).

the provider's infrastructure using enclave protection. Ryoan's goal is to prevent such covert channels and stop an untrusted application from intentionally and covertly using users' data to modulate events like system call arguments or I/O traffic statistics, which are visible to the infrastructure.

An untrusted application in Ryoan is confined by a trusted sandbox. For the Ryoan prototype we chose Native Client (NaCl) [64, 74], a state of the art user-level sandbox, as our basis (it can be built as a standalone binary, independent from the browser). NaCl uses compiler-based techniques to confine untrusted code rather then relying on address space separation, a property necessary to be compatible with SGX enclaves. The Ryoan sandbox safeguards secrets by controlling explicit I/O channels, as well as covert channels such as system call traces and data sizes.

The Ryoan prototype uses SGX to provide hardware enclaves. Each SGX enclave contains a NaCl sandbox instance that loads and executes untrusted modules. The NaCl instances communicate with each other to form a distributed sandbox that enforces strong privacy guarantees for all participating parties—the users and different service providers. Ryoan provides taint labels (similar to secrecy labels from DIFC [57]) defined by users and service providers, which allow them to ensure that any module that processes their secrets is confined by Ryoan. Confining untrusted code [47] is a longstanding problem that remains technically challenging, but Ryoan benefits from hardware-supported enclave protection. Also, Ryoan assumes a request-oriented data model, where confined modules only process input once and cannot read or write persistent storage after they receive the input. This model limits Ryoan's applicability to request-oriented server applications—but such servers are the most common way to bring scalable services to large numbers of users.

Ryoan's security goal is simple: prevent leakage of secret data. However, confining services over which the user has no control is challenging without a centralized trusted platform. We make the following contributions:

- A new execution model that allows mutually distrustful parties to process sensitive data in a distributed fashion on untrusted infrastructure.
- The design and implementation of a prototype distributed sandbox that confines untrusted code modules (possibly on different machines) and enforces I/O policies that prevent leakage of secrets.
- Several case studies of real-world application scenarios to demonstrate how they benefit from the secrecy guarantees of Ryoan, including an image processing system, an email spam/virus filter, a personal health analysis tool, and a machine translator.
- Evaluation of the performance characteristics of our prototype by measuring the execution overheads of each of its building blocks: the SGX enclave, confinement, and checkpoint/rollback. The evaluation is based on both SGX hardware and simulation.

2. Background and threat model

We assume a processor with hardware-protected enclaves, e.g., Intel's SGX-enabled Skylake (or later) architecture. SGX provides a cryptographic hash of code and initial data (called a measurement), allowing a program running in a protected enclave to verify code and data integrity and giving it access to private data encrypted by keys that the host software does not know and cannot find out. The address space of a protected enclave has its privacy and integrity guaranteed by hardware. Hardware encrypts and hashes memory contents when it moves off chip, protecting the contents from other users and also from the platform's privileged software (operating system and hypervisor). Code within an enclave can manipulate user secrets without fear of divulging them to the underlying execution platform. Code within an enclave cannot have its code or control manipulated by the platform's privileged software.

SGX's security guarantees are ideal for Ryoan's distributed NaCl-based sandbox. The sandbox confines the code it loads ensuring that the code cannot leak secrets via storage, network or other channels provided by the underlying platform. Ryoan instances communicate with each other using secure TLS connections. By collecting SGX measurements and by providing trusted initialization code, Ryoan can demonstrate to the user that their processing topology has been set up correctly.

2.1 Threat model

We consider multiple, mutually distrustful parties involved in data-processing services. A service provider is not trusted by the users of the service to keep data secret; if the service provider outsources part of the computation to other services, it becomes a user of them and does not trust them to provide secrecy, either. Each service provider can deploy its software on its own computational platform, or use a third-party cloud platform that is mutually distrustful of all service providers. We assume that users and providers trust their own code and platform, but do not trust each other's code or platforms. Everyone must trust Ryoan and SGX.

A service provider might be the same as its computational platform provider, and the two might collude to steal secrets from their input data. Besides directly communicating data, untrusted code may use covert channels via *software interfaces*, such as syscall sequences and arguments, to communicate bits from the user's input to the platform.

A user of a service does not trust the software at any privilege level in the computational platform. For example, the attacker could be the machine's owner and operator, she could be a curious or even malicious administrator; she could be an invader who has taken control of the operating system and/or hypervisor; she might own a virtual machine physically co-located with the VM being attacked; she could even be a developer of the untrusted application or OS, and write code to directly record user input.

Ryoan takes no steps to prevent each party from leaking its own secrets intentionally (or via bugs). This model is suited for the case where the service provider deploys code on its own computational platform (see §4.2 for more discussion). When executing on a different platform provider, Ryoan provides protections against a malicious OS, e.g., system call validation to prevent Iago attacks [28] (similar to Haven [21], Inktag [40], and Sego [46]) and encryption to protect data secrecy. Orthogonal techniques [27, 31, 42, 61, 78] may be used to mitigate software bugs that unintentionally leak secret input data to a computation's output. Similarly, we assume a computational platform provider is responsible for protecting its own secrets (e.g., the administrator's password).

Denial of service is outside of the scope of our threat model. Untrusted applications can simply refuse to run or the underlying untrusted operating system can simply refuse to schedule our code.

Although we consider covert channels based on software interfaces like system calls, in this paper we do not consider side or covert channels based on *hardware limitations* (§2.3) or *execution time*. Untrusted enclaves can leak bits by modulating their cache accesses, page accesses, execution time, etc. Such channels are themselves technically difficult and often require dedicated systems to address adequately [33, 35, 44, 49, 80]. Many well-regarded secure system designs factor out side/covert channels based on hardware limitations or execution time, at least to some degree [21, 52, 60, 69, 76], because doing so enables progress in designing and building secure systems. While we do not claim to prevent the execution-time channel, Ryoan does limit the use of this channel to once per request (§5.2).

2.2 Intel Software Guard Extensions

Software Guard Extensions (SGX), which is available in new Intel processors, allow processes to shield part of their address space from privileged software. Processes on an SGX-capable machine may construct an *enclave* which is an address region whose contents are protected from all software outside of the enclave (via encryption and hashing). Code and data loaded into enclaves, therefore, can operate on secret data without fear of unintentional disclosure to the platform. These guarantees are provided by the hardware [15].

SGX provides attestations of enclave identities. For our purposes it is enough to think of an enclave identity as a hash of the enclave's initial state, i.e. valid memory contents, permissions, and relative position in the enclave. Our trust of the hardware extends to these identities; particularly we assume that the initial state of an enclave cannot be impersonated under standard cryptographic assumptions. Ryoan uses SGX to attest that all enclaves have the same initial state and thus the same identity. Before passing sensitive data to Ryoan a user will request an attestation from SGX and verify that the identity is the Ryoan identity.

Knowing the initial state of an enclave ensures that Ryoan instances are not compromised. SGX restricts enclave entry to special offsets defined in the enclave preventing return-to-libc [26, 34] style attacks.

Enclave code may access any part of the address space which does not belong to another enclave. Enclave code does not, however, have access to all x86 features. All enclave code is unprivileged (ring 3), and any instruction that would raise its privilege results in a fault.

2.3 Hardware security limitations

We discuss some known security limitations in modern Intel processors. We believe these limitations must be addressed independently from Ryoan, and we hope they will be. Each of these limitations compromise Ryoan's security goals. If there are others, they also must be addressed independently from Ryoan.

SGX page faults. As currently defined, privileged software can manipulate the page tables of an enclave to observe a page-granularity trace of its code and data. Devastating attacks have been demonstrated where application-level information is used to recreate fine-grained secrets from these coarse addresses, e.g., words in a document and images [71]. If SGX enclaves serviced their own page faults, this leakage channel would disappear.

Cache timing. Two processes resident on the same core can use cache timing to obtain fine-grained information about each other. For instance Zhang et al. demonstrated (on an Amazon EC2 like plaform) the extraction of ElGamal keys from a non-colluding VM [81]. The problem is worse when processes can collude; others have demonstrated high-bandwidth covert channels using cache behavior [70, 73]. There are hardware proposals to address cache timing attacks [51].

Address bus monitoring. Although SGX encrypts data in RAM, if an attacker monitors the address bus via a sniffer or a modified RAM chip, it forms a cacheline-granularity side or covert channel. Ryoan cannot prevent such attacks without new architectural changes.

Processor monitoring. Processor monitoring units (PMUs) provide extensive performance counter information for on-chip events. If the PMU is updated about events that occur in enclave-protected execution, the operating system could use the information as a covert channel to learn secrets via untrusted code which could modulate

Module property	Enforce	Reason
OS cannot access module memory (§2.2).	SGX	Security
Initial module code and data verified (§2.2).	SGX	Security
Can only address module memory (§2.4).	NaCl	Security
Ryoan intercepts syscalls (§2.4,§4.3).	NaCl	Security
Cannot modify SGX state (§5).	NaCl	Security
User defines topology (§4.1).	Ryoan	Security
Data flow tracked by labels (§4.2).	Ryoan	Security
Memory cleaned between requests (§5).	Ryoan	Security
Module defines initialized state (§5.4).	Ryoan	Perf.
Unconfined initialization (§5).	Ryoan	Compat.
In-memory POSIX API (§5.1)	Ryoan	Compat.

Table 1: Properties Ryoan imposes on untrusted modules, the technology that enforces them, and the reason Ryoan imposes them.

its behavior to e.g., inflate certain event counts.

According to measurements on Skylake processors, certain monitoring facilities are turned off during enclave execution (e.g., Precise Event Based Sampling (PEBS)), however the uncore counters (e.g., cache misses, TLB misses) are enabled [32]. It is unknown at this time how effective attacks based on processor monitoring will be.

Part of the purpose in constructing the Ryoan prototype is to demonstrate the importance of addressing these hardware-based information leaks.

2.4 Native Client

Google Native Client (NaCl), is a sandbox for running x86/x86-64 native code (a NaCl module) using software fault isolation. NaCl consists of a verifier and a service runtime. To guarantee that the untrusted module cannot break out of NaCl's SFI sandbox, the verifier disassembles the binary and validates the disassembled instructions as being safe to execute.

NaCl executes system calls on behalf of the loaded application. System calls in the application transfer control to the NaCl runtime which determines the proper action. Ryoan cannot allow the application to use its system calls to pass information to the underlying operating system. For example, if Ryoan passed read system calls from the application directly to the platform, the application could use the size and number of the calls to encode information about the secret data it is processing. We discuss the details of the confinement provided by Ryoan in Section 5.1.

3. Design overview

Ryoan is a distributed sandbox that executes a directed acyclic graph (DAG) of communicating untrusted modules which operate on sensitive data. Ryoan's primary job is to prevent the modules from communicating any of the sensitive data outside the confines of the system (including external hosts and the platform's privileged software).

Ryoan prevents modules from leaking sensitive data by decoupling externally visible behaviors from the content of secret data. SGX hardware limits externally visible behaviors to explicit stores to unprotected memory and

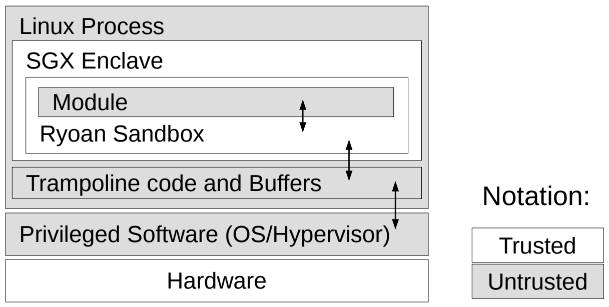


Figure 1: A single instance of Ryoan's distributed sandbox. The privileged software includes an operating system and an optional hypervisor.

use of system services (syscalls). Unprotected stores are eliminated by the NaCl tool chain and run time. Ryoan mostly eliminates system calls by providing their functionality from within NaCl. For example, Ryoan provides `mmap` functionality by managing a fixed-sized memory pool within the SGX enclave. However, untrusted modules must read input and write output so Ryoan provides a restricted IO model that prevents data leaks (e.g., the output size is a fixed function of input size). Table 1 summarizes the properties Ryoan imposes on untrusted code to achieve secure decoupling of observable behavior from secret input data.

Figure 1 shows a single instance of the Ryoan distributed sandbox. A principal (e.g., a company providing software as a service) can contribute a module which Ryoan loads and confines, enabling the module to safely operate on secret data. A module consists of code, initialized data, and the maximum size of dynamically allocated memory. The NaCl sandbox uses a load time code validator to ensure that the module cannot violate the sandbox by reaching outside of its address range or making syscalls without Ryoan intervention.

For backward compatibility, Ryoan modules support programs written for `libc`, which could include fully compiled languages and runtimes built on top of `libc`. To reduce memory use, our Ryoan prototype does not support a just-in-time compiler (JIT), though NaCl supports it [17]. Ring 0 execution is disallowed in enclaves so Ryoan cannot directly support an operating system or hypervisor. A Ryoan module can be a Linux program, or it could contain a library operating system [21].

Ryoan does not trust other software on the computational platform, including privileged software, i.e., operating system and hypervisor. Instead, Ryoan assures its own secrecy and integrity by executing in a hardware-protected enclave. Hardware attests to Ryoan's initial state becoming the anchor for Ryoan's chain of trust (Figure 2). SGX generates an unforgeable remote attestation for the user that an Ryoan instance is executing in an enclave on the platform. The user can establish an encrypted channel that she knows terminates within that Ryoan instance. SGX guarantees the enclave cryptographic secrecy and integrity against manipulation by privileged software.

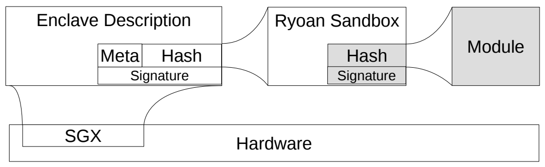


Figure 2: The Ryoan chain of trust. SGX hardware attests that a valid instance of Ryoan is executing (Hash) with an intended SGX configuration (Meta). Ryoan ensures that the expected binary is loaded with a signed hash from the software provider (grey).

A master enclave creates all Ryoan instances and they establish cryptographically protected communication channels among themselves as specified by the user. Once the distributed topology has been established, the master forwards the attestations for each node in the topology to the user who verifies that the configuration matches her specification. Then the user inputs her secret data. Ryoan provides simple labels to protect secret data added by modules in the DAG. All of Ryoan's instances together form a distributed sandbox that protects secret input data from being leaked by the untrusted code modules that operate on it.

Ryoan identity and module identity. SGX attests to the Ryoan sandbox using processor hardware and the Ryoan sandbox attests to the module's initial state (Figure 2) using software cryptography. SGX supports two forms of identity, one based on a hash of the module's initial state (MRENCLAVE) and one based on a public key, product identifier and security version number (MRSIGNER). SGX can verify Ryoan using either form of identity; our prototype uses MRENCLAVE. Ryoan can support software analogs of either identity for untrusted modules; the prototype identifies modules by the public key that signs them.

In the next section, we will describe Ryoan's distributed properties and how they are enforced, followed by a more detailed explanation of how individual instances confine modules.

4. The Ryoan distributed sandbox

The Ryoan sandbox is distributed, with different instances confining untrusted modules while all instances communicate to enforce global properties like the communication topology and secrecy labels for code and data.

4.1 Enforcing Topology

The user either defines the communication topology of confined modules or explicitly approves it. A topology is a DAG of modules with unidirectional links (see §5.2 for why Ryoan requires a DAG). The DAG specification is first passed to an initial enclave which we call the *master*. It contains standard, trusted initialization code provided by Ryoan. The master requests that the operating system start enclaves that contain Ryoan instances for modules listed in the specification. These enclaves can be hosted on different machines. The master uses SGX to perform local or remote attestation to verify the validity of individual Ryoan enclaves, then lets neighbor enclaves in the DAG establish cryptographically protected communication channels via key exchange using the untrusted network or local inter-process communication as a transport. The user can verify the validity of the master via attestation, and ask it whether a desired topology has been initialized. After that, the user establishes secure channels with the entry and exit enclaves of the DAG, and starts data processing.

The master enclave is convenient but not essential to our design. We could instead append a DAG specification to each user request, and have each enclave verify the identities of its neighbors according to the specification before sending its output.

Figure 3 shows an example of Ryoan processing input from user Alice whose sensitive data is processed by both 23andMe and Amazon. Each Ryoan instance executes in an enclave on the same or different machines. The host machine(s) might be provided by 23andMe, Amazon, or a third party. In all cases, Ryoan assures no leakage of the user's secrets and also prevents leakage of any trade secrets used by 23andMe and Amazon.

4.2 Label-based model for communication

Ryoan labels. Ryoan adapts previous label-based systems to enable multiple mutually distrustful modules to cooperate on sensitive data. Ryoan uses secrecy labels to mark secret data and enclaves which have seen that secret data. Ryoan's labels are similar to a DIFC system [45, 52, 60, 69, 76], but far simpler. Ryoan labels could also be thought of as taint tracking [30] at enclave-level granularity, with per-principal classes of taint. Taint is attached to data at unit of work granularity (where units of work are application defined). Conceptually, *a label is a set of tags*, where each tag is an opaque identifier drawn from a large universe that identifies a principal, indicating secrets from this principal.

In our prototype, a user's tag is his/her public key. A company can use its private key to sign its module binaries, and use the associated public key as the tag for those modules. The company can also use different key pairs to sign its module binaries to make them different principals, enabling privilege separation.

Label manipulation rules. Each module has the ability to add or remove a single tag that corresponds to its principal — each module can declassify its own secrets. When a module reads data with a non-empty label (e.g., from a user or another module's output), it merges the data's label with its current label which becomes the new label for both the module and the data. Ryoan marks a module's output data with the module's label.

In Figure 3, input from Alice is labeled with her tag, and the first 23andMe module adds the 23andMe tag, to make sure that its secrets cannot flow back to the user after

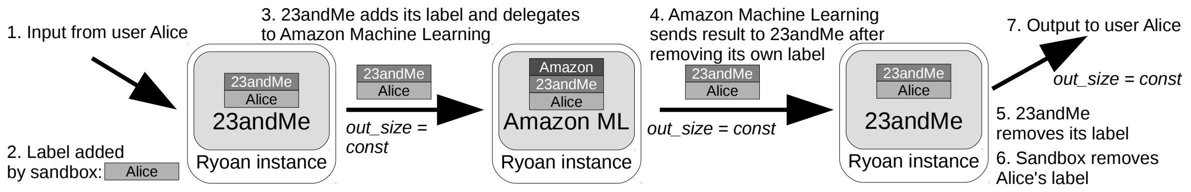

1. Input from user Alice
2. Label added by sandbox: Alice
23andMe
Alice
23andMe
Ryoan instance
3. 23andMe adds its label and delegates to Amazon Machine Learning
23andMe
Alice
out_size = const
Amazon
23andMe
Alice
Amazon ML
Ryoan instance
4. Amazon Machine Learning sends result to 23andMe after removing its own label
23andMe
Alice
out_size = const
23andMe
Alice
23andMe
Ryoan instance
5. 23andMe removes its label
6. Sandbox removes Alice's label
7. Output to user Alice
out_size = const

Figure 3: Ryoan's distributed sandbox. Ryoan instances manage labels on data and modules. The user's tag is propagated to all modules, making them confined after receiving input; For example, 23andMe's tag is kept when it outsources to Amazon Machine Learning to prevent leaking secrets from 23andMe.

handing them off to Amazon's machine learning module. This control is important since the user is in control of the topology. The second 23andMe module removes its tag from its output's data label. In a sense, the public key of 23andMe creates a group and both of its modules are members of the group—verified by Ryoan because both are signed with that key. Ryoan is trusted to remove the user's tag when it communicates over a protected and authenticated connection to the user.

4.2.1 Non-confining labels

A Ryoan instance is created with an empty label, and the module can add the tag that corresponds to its principal at any time. If the label does not contain tags from other principals, the module is not confined and may perform any file system operation, network communication, or address space modification permitted by Ryoan and NaCl. For example, it can freely initialize its state by reading from the network or file system. Ryoan allows unfettered access to external resources because the principal's own tag means that the module may have seen secrets only from itself. In Ryoan's threat model, principals trust their own module not to leak its own secrets (§2.1).

In many DIFC systems [45, 52, 60, 69, 76], principals are independent from the application code, e.g., multiple users (principals) use the same wiki Web application, and the users do not trust the application. Ryoan allows application owners (service providers) to be principals who trust their own code, which is different from the standard DIFC model. Although a service provider's code may have bugs that cause it to release its own secrets in its output, that is not within the threat model for Ryoan and can be mitigated using orthogonal techniques (§2.1). Ryoan protects a principal's data when that data is processed by modules that are not under control of the principal.

A service provider can host its modules and secret data on its own machines to protect them. However, if it chooses to use a third-party computational platform that it does not trust, its modules containing non-confining labels need encryption to protect persistent secrets from the platform. Ryoan uses the SGX sealing feature to store secret data on behalf of modules. Sealing provides an encryption key only accessible to enclaves with the same identity executing on the same processor. For Ryoan, all enclaves are Ryoan instances and have the same identity. Any data that the module wants to persist securely is passed to Ryoan, which adds its own metadata, including the public key of the requesting module. Ryoan seals the data and metadata and writes the result into a file. The metadata allows Ryoan to persist data on behalf of different modules and allows it to restrict any module's access to its own data.

4.2.2 Confining labels

When a module's label contains tags of other principals (as a result of receiving secrets from a user or another module's output), it enters a confined environment strictly enforced by Ryoan. Ryoan must prevent confined modules from leaking data that belong to other principals. Such labels are called *confining labels*.

Modules with confining labels are disallowed to persist data. As a result, Ryoan's label system is far simpler than DIFC systems [45, 57, 60, 69, 76]. Confined modules have seen secret data from other principals, so allowing them persistent storage violates Ryoan's "one-shot" request-oriented data model—a module processes a request's data once and only once.

4.2.3 Ryoan data audit trail

As data traverses the DAG of modules, Ryoan tracks which modules process each piece of user work. The audit trail for each work unit is available to the user as part of a DAG's output. While Ryoan cannot verify that modules are performing their intended or claimed function, an audit trail can still be useful. For example, a given piece of data might have been processed by a version of a module which is known to be faulty. Whether a user wants the audit trail and for what purpose is dependent on the application and the user.

4.3 Data oblivious communication

One of the primary safety functions of Ryoan is to prevent the computational platform from inferring secrets about the input data by observing data flow among modules. Therefore, data flow must be independent from the contents of the input data: Ryoan never moves data in response to activity under the control of the untrusted module once the module has read its input data. This safety property is sometimes called being data oblivious [58].

Units of work can be any size, but Ryoan ensures that data flow patterns do not leak secrets from input data by making module output size a fixed, application-defined function of the input size. Ryoan protects communication with the following rules: (1) each Ryoan instance reads its entire input from every input-connected Ryoan instance before the module starts processing. (2) the size of the output is a polynomial function of the input size, specified as part of the DAG. Ryoan pads/truncates all outputs to the exact length determined by the polynomial and the size of the input. (3) Ryoan is notified by the module when its output is complete , and it writes the module's output to all output-connected Ryoan instances. Ryoan encapsulates module output in a message that contains metadata which describes what is module output and what is padding (if any). The metadata is interpreted, and any padding is stripped away by the next Ryoan instance before exposing the data to its module. These rules are sufficient because they ensure that output traffic is independent from input data (though there are possible alternatives, for example, each request could specify its output size).

Consider the scenario in Figure 3. Each input comes from a user. The user can choose to leak the size of the input, or she can hide the size by padding the input. The description of the DAG specifies that (1) the output of 23andMe's first module is padded to a fixed size defined by 23andMe which can hold the largest possible user input, (2) the output of Amazon Machine Learning's classifier module is padded to a fixed size to encode the classification result, and (3) the response to the user from 23andMe's second module is also padded to a fixed size that can hold the largest possible result. Each Ryoan instance must receive the complete input of a work unit before executing its module.

Ryoan ensures that output size is a fixed function of the input, so it is a module's mistake if the output is not large enough. Ryoan will truncate outputs that are too large and pad outputs that are too small. However, a module author should be able to describe the maximum possible output for a given sized input request. For example, a spam detector's output will be the size of the input mail message (which is just copied) plus a constant size sufficient to hold the spam rating for the email.

5. Module confinement

Ryoan relies on instances of its sandbox to prevent modules from leaking sensitive data to an adversary, including the platform's privileged software. To that end, a sandbox instance enforces the life cycle, system service restrictions, and input-output behavior of the module.

Module validation Ryoan module validation ensures that modules are safe to execute by enforcing a set of constraints on the code being loaded. Ryoan uses NaCl's load-time code validator to ensure that the module's code adheres to a strict format. NaCl's code format is designed to be efficiently verified and efficiently sandboxed, restricting control flow targets and cleanly separating code from data. Memory accesses are confined to remain within the address space occupied by the module, including fetches for execution. The detailed guarantees of NaCl are available as prior work [64, 74] and Ryoan does not change the base guarantees of the NaCl sandbox. Ryoan adds the constraints that modules may not contain any SGX instructions, and that control flow is constrained to the initial module code; i.e., Ryoan disallows dynamic code generation.

Module life cycle A Ryoan instance enforces the following life cycle on its module: creation, initialization, wait, process, output, destruction/reset. The sandbox begins by validating its module and verifying that its identity matches the DAG specification. It allows the module to initialize with an empty label and the module can give itself a non-confining label (§4.2.1). In both cases the module has full access to the system services exposed by vanilla NaCl. Non-confined initialization makes module creation more efficient and it makes porting easier because initialization code can remain unchanged.

Modules signal Ryoan when initialization is complete by calling `wait_for_work`, a routine implemented by Ryoan. Once a module is initialized, it processes a request, generates its output, and then is destroyed or reset to prevent accumulating secret data. Ryoan instances are request oriented: input can be any size, but each input is an application-defined "unit of work." For example, a unit of work can be an email when classifying spam, or a complete file when scanning for viruses. Each module gets a single opportunity to process its input data.

5.1 Ryoan's confined environment

Any module with a confining label is executed in Ryoan's confined environment. Ryoan's confined environment is intended to prevent information leakage while reducing porting effort. When a module receives the secret data contained within a request, it enters the confined environment and loses the ability to communicate with the untrusted OS via any system call. Therefore, Ryoan must provide a system API sufficient for most legacy code to perform their function. Ryoan provides these services.

- The most important service is an *in-memory virtual file system*. First, Ryoan allows files to be preloaded in memory, and the list of preloaded files must be determined before the module is confined; e.g., they can be listed in the DAG specification, or requested by the module during initialization. Ryoan presents POSIX-compatible APIs to access preloaded files that are available even after the module is confined. Second, a confined module can create temporary files and directories (which Ryoan keeps in enclave memory). When the module is destroyed or reset, all temporary files and directories are destroyed, and all changes to preloaded files are reverted.
- `mmap` calls are essential to satisfy dynamic memory allocation, so Ryoan supports anonymous memory map-

pings by returning addresses from a pre-allocated memory region. The maximum size of the region must be decided before the module becomes confined.

Ryoan's confined environment is sufficient for many data-processing tasks. For example, ClamAV, a popular virus scanning tool loads the entire virus database during initialization; when scanning the input such as a PDF file, it creates temporary files to store objects extracted from the PDF. Ryoan's in-memory file system satisfies these requirements.

However, if an application needs a large database that does not fit in memory when processing data, Ryoan cannot support it as a single module. A workaround would be to partition the database, and use multiple modules to load different partitions and perform different parts of the task, if that is feasible for a particular application.

Any design alternative that allows access to persistent files (as opposed to Ryoan's in-memory files) must cope with the covert channel created by allowing the OS to see file reads, which might occur based on computation within the untrusted module. Ryoan eliminates this channel by executing from memory only. All Ryoan modules must fit into memory for their entire lifetime because any "swapping" done by Ryoan will create a covert channel between the module and the operating system. File access techniques based on oblivious RAM (ORAM [50, 62]) can hide data access patterns, but at a performance and resource cost we deem too high for Ryoan.

5.2 Processing-time channels

A confined module cannot communicate with the untrusted OS via system calls, but it determines when its data processing is finished, which can be a channel to the OS to leak secrets by choosing different processing times depending on the data.

Fixed processing time. Timing channels can be eliminated by forcing a fixed processing time whose length is determined before the module has seen any data. The OS cannot directly determine when the module completes, so the Ryoan runtime can pad execution time by busy waiting. However, controlling its timing without the cooperation of the operating system is a challenge. Fixed processing time can be quite expensive for computations with widely variable run times, because all runtime would be padded to the worst case. However, fixed processing time can be quite modest for computations with highly predictable run times (e.g., evaluating certain machine learning models like decision trees) or with light throughput requirements. Fixed time execution does not leak information, though we defer to future work building a Ryoan instance that supports it. To add some flexibility with no loss of security, execution time could also be a fixed function of input length.

Quantized processing time. Processing time channels are mitigated by reducing the granularity of potential processing times by padding execution to a fixed number of quantized, pre-defined values [20, 67, 79, 80]. Because Ryoan only allows modules to see sensitive data once, individual modules can only leak a number of bits that is proportional to the logarithm of the number of distinct execution durations (e.g., if the code terminates after one of eight different statically determined intervals, it leaks three bits).

Randomness. Users can specify whether confined modules need access to randomness. If the user allows, a module can access randomness via the processor, e.g., Intel's RDRAND instruction. Ryoan does not allow confined modules to get randomness from the operating system. Access to randomness means a malicious module can leak random bits from an input, for example choosing an input bit at random and leaking it using its processing time. If the user repeats input data, a malicious module with access to randomness can eventually leak the entire input over its processing-time channel, even though it only leaks once for each input unit of work. Using a fixed processing time eliminates this channel.

One shot at input data. Ryoan is designed to allow each module a single opportunity to process its input data, with no opportunity to carry forward state from one input to the next. This one-shot policy limits data leakage. Therefore, Ryoan must prevent a module from accessing the same input again after reset. Ryoan enforces the one-shot policy by 1) requiring that the data processing topology is a DAG to avoid cycles; 2) Ryoan's reset mechanism deletes all data dependent on state modified after secret data is read; 3) Ryoan prevents input replay attacks by reinitializing all secure connections if any connection is ever broken. Secure communication protocols contain protection against replay attacks [75], so reinitializing broken links prevents input replay. Note that the OS can pause or stop the execution of an SGX enclave, but it cannot rollback its state [15], which means the state of a secure connection cannot be rolled back. Ryoan itself uses high-quality randomness available via the processors RDRAND instruction to establish secure connections, which does not rely on the OS.

5.3 Protecting Ryoan from privileged software

A Ryoan instance requires services provided by the untrusted operating system and possibly the hypervisor. The Ryoan instance must check the results coming from the untrusted operating system to make sure the OS is not being misleading. Most of these checks can be transparently inserted into `libc`, the lowest level of software that communicates with the operating system. Ryoan-libc is Ryoan's replacement for `libc` and it manages system call arguments and checks their return values. The Ryoan sandbox code invokes Ryoan-libc through standard `libc` functions, such as the wrappers for system calls (e.g., `read`).

Iago attacks. Ryoan-libc guards against all known Iago attacks [28] by keeping state in enclave memory and carefully checking the results of system calls e.g., making sure that addresses returned from `mmap` do not overlap with previously allocated memory (like the stack). For Linux, the system call interface can be secured e.g., by maintaining semaphore counts in enclave memory and duplicating futex [37] memory inside and outside the enclave. Ryoan shares the need for this type of checking with all systems distrustful of the operating system [29, 40, 46], though some check at a lower level than system calls [21].

Controlling an enclave's address space. SGX provides user control of memory mapping, including permissions. Ryoan-libc maintains a data structure that is equivalent to the kernel's list of virtual memory areas (VMAs). It knows about each mapped region and its permissions. Map requests are fulfilled eagerly and verified by Ryoan-libc at the time of the request (i.e., as part of the `mmap` call), not at page fault time.

SGX dictates a very specific procedure for verifying enclave mappings. A typical new mapping proceeds as follows: (1) Untrusted code notifies the kernel of a new desired mapping via a system call made by Ryoan-libc. (2) The kernel selects new enclave page frames to satisfy the mapping and modifies the page tables to map the frames at the requested virtual address with the requested permissions. (3) Untrusted user code resumes and passes control to enclave code. (4) Enclave code verifies that the mapping completed as expected by invoking the SGX instruction `EACCEPT` on every new page. The `EACCECPT` instruction accepts a virtual address and protection bits and verifies that the current address space maps that page to a valid, SGX protected 4KB physical frame. New pages added to the enclave always start out with read and write permissions and their contents are zeroed by hardware.

If the user wants something other than read and write permission, SGX provides the `EMODPE` instruction to make them more permissive and the `EMODPR` instruction which makes them less permissive. `EMODPE` is only available to enclave code while `EMODPR` is only available to privileged software (ring 0, outside of the enclave). If an enclave desires less permissive page access rights, it must signal privileged software to request the restriction, but can validate that it was done correctly through another use of the `EACCEPT` instruction. Note that the OS can always restrict page permissions against the enclave's wishes, which will create more permission faults.

Ryoan-libc emulates `mmap` behavior by doing work required by SGX on behalf of the user. For instance, if the user expects new pages to have particular contents (e.g., she privately mapped a file) and to be read-only, Ryoan-libc can copy the requested portion of the file into enclave memory and ensure those pages have read-only permissions before returning.

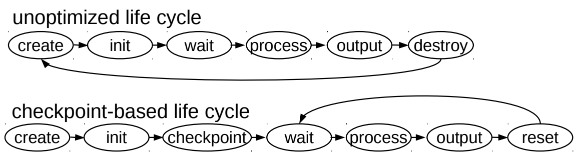


Figure 4: Instance life cycle: unoptimized vs checkpoint-based.

5.4 Optimizing module reset

The restrictions necessary to confine modules create execution time and memory space overheads. In this section we discuss strategies for mitigating these overheads.

Checkpoint-based enclave reset. Creating and initializing modules often requires far more CPU time than processing a single request (see Section 8 for measurements). For instance, loading the data necessary for virus scanning takes 24 seconds; orders of magnitude greater than the ≈0.124 seconds it takes to process a single email. Ryoan manages the module life cycle efficiently using checkpoint-based enclave reset.

Creating and initializing a hardware protected enclave is slow (e.g., we measured 30 ms for a small enclave). Compounding the problem is that applications often do not optimize their own initialization sequence on the assumption that it is not frequently executed. But Ryoan does not allow any data from one input to be carried forward to the next, so each input requires that computation begins from the same, non-secret state, making initialization a bottleneck.

Ryoan provides a checkpoint service that allows the application to be rolled back to an untainted, but initialized, memory state (Figure 4). In our prototype this state is at the first invocation of `wait_for_work`. Ryoan does not allow an enclave that has seen secret input to be checkpointed, because its data model is request-oriented: modules should not depend on past requests to operate. Checkpointing a module that has seen secret data would (potentially) give that module multiple execution opportunities on a single request's data.

Checkpoint restore allows Ryoan to save the cost of tearing down and rebuilding the SGX enclave and it saves the cost of executing the application's initialization code. Ryoan checkpoints are taken once, but restored after each request is processed. Therefore, Ryoan makes a full copy of the module's writeable state and simply tracks which pages get modified (avoiding a memory copy during processing). Only the contents of pages that were modified during input processing are restored (§6.6). SGX provides a way for enclave code to verify page permissions and be reliably notified about memory faults, which is necessary to track which pages are written.

Batch requests before reset. A user might want more efficiency by allowing a module to process several input units of work before it is reset. Whether batching multiple inputs within a single request constitutes a threat is user and application dependent. But if a module can process more than one unit for the same data source, it can accumulate secrets across two wait-process-output cycles. Having access to more secret data for longer exacerbates the problem of slow leaks (e.g., timing channel leaks). For example, an email-filtering module allowed to process multiple emails without resetting could leak a password contained in one email by using the processing-time channel across multiple wait-process-output cycles.

6. Implementation

The Ryoan instance prototype is based on NaCl version 2d5bba1 with the last upstream commit on Jan 19 2016. We leverage NaCl's existing sandboxing guarantees to control the module's access to the platform. NaCl ensures that the module in the sandbox has no direct access to OS services. We ported NaCl for use in SGX with the introduction of the Ryoan-libc layer. NaCl depends on libc to interface with the platform. Ryoan-libc makes system calls on behalf of a Ryoan instance after checking that the system call is allowed. We modified eglibc's dynamic linker to support loading Ryoan into enclaves, but all modules must be statically linked. We base Ryoan-libc on eglibc 2.19 which is compatible with our version of NaCl.

6.1 Constraints of current hardware

Ryoan relies on features from version 2 of the SGX hardware, while only version 1 is currently available. Version 2 adds the ability to modify enclaves dynamically, i.e., augmenting an executing enclave with new memory and changing protections on existing enclave memory. Furthermore, our first generation SGX-capable machine makes only a limited amount of physical memory available to SGX (128MB on our machine).

6.2 Ryoan-libc

Ryoan-libc manages interactions with the untrusted operating system. It is impossible for the OS to read enclave memory; so Ryoan-libc marshals system call arguments into the process' untrusted memory and copies back results. Interposition from libc is common for applications that do not trust the operating system [29, 40, 46], while Haven protects a smaller system interface [21].

Fault handling. Signals allow user-level code to be interrupted by the system. The source of most signals is unreliable when the OS is untrusted, but SGX allows us to get reliable information about memory faults; this allows Ryoan-libc to expose this information to Ryoan instances though the normal signal handler registration interface. Ryoan-libc signal support is currently limited to the memory fault signal (SIGSEGV).

After any fault, exception, or interrupt the OS returns control to untrusted trampoline code contained within the process. In the case of a memory fault, rather than simply resuming the enclave where it was paused (as in the normal case), our trampoline code enters the enclave where it can read reliable information about the fault from SGX and make necessary arrangements to fix the fault (e.g., change permissions). After handling the fault, the enclave exits and then our trampoline resumes the enclave at the instruction that caused the memory fault. We cannot protect the trampoline code from the OS, but it can only enter the enclave using the EENTER instruction, which will transfer control to our fault-checking entry point, or resume the enclave using the ERESUME instruction which will re-execute the instruction that faulted. If the enclave is resumed without calling the enclave fault handler, the instruction will simply refault.

6.3 Module address space

x86-64 NaCl allocates a 84 GB region for a NaCl module with 4 GB of module address space flanked above and below by 40 GB of inaccessible guard pages, but current SGX hardware only allows enclaves with 64 GB of virtual address space. Fortunately, the original x86-64 NaCl design [64] overestimated the amount of guard pages needed to allow for future changes in the architecture. A detailed analysis [6] indicates we can remain safe by keeping the upper guard region unchanged but reducing the lower region from 40 GB to 4 GB. A Ryoan instance therefore requires 48 GB of virtual address space which fits into current SGX hardware.

6.4 I/O control

A Ryoan instance controls its module's access to files and request (work unit) buffers when it is confined, preventing the module from leaking data via direct syscalls.

In-memory virtual file system. A confined module cannot access the file system, but Ryoan implements POSIX-compatible APIs for in-memory virtual files, including preloaded files and temporary files. An in-memory file is backed by a set of 4 KB blocks that are indexed by a two-level tree structure (similar to a page table). The blocks of a file are allocated on demand as the file grows. The maximum size of an in-memory file is 1 GB. An in-memory directory is backed by a hash table, and we use reference counts to track the lifetime of files. This virtual file system supports standard APIs including `open`, `close`, `read`, `write`, `stat`, `lseek`, `unlink`, `mkdir`, `rmdir` and `getdents`. When the module writes a preloaded file, the Ryoan instance keeps the original file blocks. When the module is reset, preloaded files are restored to their original versions and temporary files are deleted.

Input/output buffers. For each unit of work, a module calls `wait_for_work` (a system service implemented by Ryoan), and the Ryoan instance reads its entire input from all input channels into memory buffers before returning to the module. After processing the work unit, the module's output is written to a buffer, and in the next `wait_for_work` call, the Ryoan instance flushes the buffer to output channels after padding or truncating the output to a size calculated using a polynomial function of input size according to the DAG specification. The module accesses these buffers via file descriptors using APIs implemented in the virtual file system, just like using regular pipes or sockets.

6.5 Key establishment between enclaves

Ryoan instances implement protected channels using an authenticated encryption algorithm (AES-GCM [55]) provided by the libsodium [9] library. Encryption keys are agreed on at runtime using Diffie-Hellman key exchange. SGX allows you to embed the key parameters in attestations, accelerating a Diffie-Hellman key exchange between enclaves [16]. On our hardware (§8), SGX key exchange takes 1.78ms while OpenSSL takes 1.90ms. Randomness is required for key exchange and Ryoan uses the x86 instruction RDRAND to obtain it.

6.6 Checkpointing confined code

Ryoan uses page permission restriction and fault information to detect module writes. Recall that SGX provides reliable memory page permissions and information about memory faults; Ryoan does not trust the OS (§6.2). The entire module is write protected by the OS when it is confined. Ryoan verifies that the protection was done using `EACCECPT`. As the module writes, the Ryoan instance catches permission faults and records the page's address before changing the permissions to allow writes and resumes the module. However, it still needs the OS to change permissions in the page table, which requires ring-0 privilege. In fact, the page fault causes an exit from the enclave; the kernel catches it and invokes the Ryoan instance's signal handler. The handler first executes outside enclave mode and makes an `mprotect` syscall to change page permissions, then enters enclave mode to update SGX page permissions with `EMODPE`. After that, the handler returns and normal execution resumes.

To reset the enclave, all written pages are restored to their initial value and made unwritable again. In our prototype, before an untrusted module is confined for the first time, the Ryoan instance creates a checkpoint by copying the module's complete writable memory state. This copy-on-initialize strategy optimizes the case where Ryoan instances are created once and then used and reset for many requests. If the copy-on-initialize cost is too high, Ryoan could incrementally create the checkpoint by doing copy-on-write for each request, gradually accumulating and preserving unmodified versions of any page modified during any execution.

In our prototype the checkpoint is taken when the module is blocking on `wait_for_work`, and restore occurs the next time the module blocks on `wait_for_work`. This gives module writers clear semantics about what state will not persist across invocations, and allows the Ryoan instance to purge any secrets kept in registers.

Restoring a checkpoint does incur additional page faults which could be used as a channel to leak data. We find these additional faults acceptable as even normal page accesses by the module are a channel between module and OS that SGX does not close [71]. Page faults will continue to leak information about enclave execution until future generations of hardware enclaves can service their own page faults (§2.3). To make Ryoan execution on current SGX hardware more secure, we could save/restore all writable regions of the module instead of tracking individual pages using write protection. This strategy is less efficient but does not leak additional per-page information.

7. Use cases

This section explains four scenarios where Ryoan provides a previously unattainable level of security for processing sensitive data. For all examples, the Ryoan instances could execute on the same platform or on different platforms, e.g., the entire computation might execute on a third-party cloud platform like Google Compute Engine, or a provider's module might execute on its own server. Ryoan's security guarantees apply to all scenarios.

7.1 Email processing

A company can use Ryoan to outsource email filtering and scanning while keeping email text secret. We consider spam filtering and virus scanning, using popular legacy applications — DSPAM 3.10.2 and ClamAV 0.98.7.

The computation DAG for this service contains four Ryoan instances, each confining a data processing module (see Figure 5). An email arrives at the entry enclave over a secure channel, which simply distributes the email text and attachments to the enclaves containing DSPAM and ClamAV, respectively. The results of virus scanning and spam filtering are sent to a final post-processing enclave which constructs a response to the user over a secure channel.

7.2 Personal health analysis

Consider a company (e.g., 23andMe) that provides customized health reports for users based on a variety of health data. 23andMe accepts a user's genetic data, medical history, and physical activity log as input, extracts important health features from these data, and predicts the likelihood of certain diseases [1]. Since genetic and health information is extremely sensitive, users may not feel comfortable with the company keeping their data. To encourage use of the service, 23andMe can deploy it with Ryoan, assuring users that the code that processes their data cannot retain or leak their secrets.

23andMe owns its research results about the associations between diseases and health features, but it may want to use a third-party cloud machine learning service

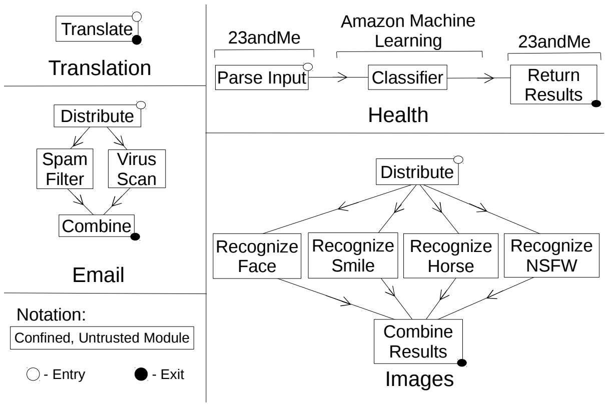


Figure 5: Topologies of Ryoan example applications. Nodes in the graph are Ryoan instances, though we identify them by their untrusted module. Users establish secure channels with trusted Ryoan code for the source and sink nodes to provide input and get output respectively.

(e.g., Amazon Machine Learning [2]), to train its model and generate predictions. 23andMe's trade secret would be how to map a user's complex, multi-modal health data onto machine learning features. Amazon Machine Learning provides a way to train models based on unlabeled features and software (a classifier) which queries that model. After training a model this way 23andMe want to keep the input to the classifier a secret from parties which have the means to map the inputs back to secret heath data: users of their service. Ryoan enables 23andMe to outsource machine learning tasks to Amazon while protecting its proprietary transformation from user data to health features.

Secrecy for both users and 23andMe is protected with a DAG shown in Figures 3 and 5. 23andMe compiles a training data set which it transmits to Amazon to construct a model. Amazon provides the classifier which queries that model as a Ryoan module. Users provide their genetic information, medical history, and activity log in a request. Upon receiving a user's request, 23andMe's first module constructs a boolean vector of health features and forwards it to Amazon's module. Amazon's module generates predictions based on the model and forwards the result to 23andMe's second enclave, which then forwards the result back to the user.

The user's label is kept throughout the entire pipeline, so that all the enclaves are confined when they receive the user's input, and prevented from leaking information about the input. Further, 23andMe keeps its label with the request sent to Amazon, so that Amazon cannot leak data about 23andMe's heath features to other parties (in particular the user), since they cannot remove 23andMe's label in order to release data out of Ryoan's confinement. Amazon's module passes the results of classification to another module owned by 23andMe which verifies its proprietary transformations are not being leaked before removing the 23andMe label and allowing results to be returned to the user.

The actual prediction model is unknown to us and out of scope for this paper; but our workload uses our knowledge of best practices. We train a support vector machine (SVM), and choose 20 well studied diseases and the top 500 genes that have correlations with the them, according to a database provided by DisGeNet [14]. The SVM models are trained using synthetic data based on that database. Our prototype uses stochastic gradient descent as the training algorithm [24] which allows incremental updates to existing models.

7.3 Image processing

Image classification as a service is an emerging area that could benefit from Ryoan's security guarantees (e.g., Clarifai [3] or IBM's Visual Recognition service [5]). We envision a scenario where a user wants different image classification services based on their expertise. For example, one service might be known for accurate identification of adult content [53] while another might do an excellent job recognizing and segmenting horses. The image processing DAG in Figure 5 shows an example where an image filtering service outsources different subtasks to different providers and then combines the results. The user's label is propagated to all processing enclaves, causing Ryoan to confine their execution. Our prototype implements all of these detection tasks using OpenCV 3.1.0, and each detection task loads a model that is specialized to the detection task and would represent a company's competitive advantage.

7.4 Translation

A company uses Ryoan to provide a machine translation service while keeping the uploaded text secret. Users upload text to the translation enclave and get the translated text back. Our prototype uses Moses [10], a statistical machine translation system. We train a phrase-based French to English model using the News Commentary data set released for the 2013 workshop in machine translation [13].

8. Evaluation

We quantify the time and space costs of Ryoan and its components by measuring the execution of the use cases described in the previous section using a combination of real hardware and emulation.

All benchmarks are measured on a Dell Inspiron 7359 laptop with Intel Core i5-6200U 2.3 GHz processor (with Skylake microarchitecture and SGX version 1) and 4 GB RAM. We use a laptop because it contains the first SGX-enabled processor we could purchase. We use Intel's SGX Linux Driver [8] and SDK [7] to measure the costs of SGX instructions. We inserted delays based on those measurements, and appropriate TLB flushes consistent with Intel's SGX specification to measure the performance of Ryoan. To test our implementation and overcome the limitations of our hardware, we built an SGX emulator based on QEMU [11] (full emulation

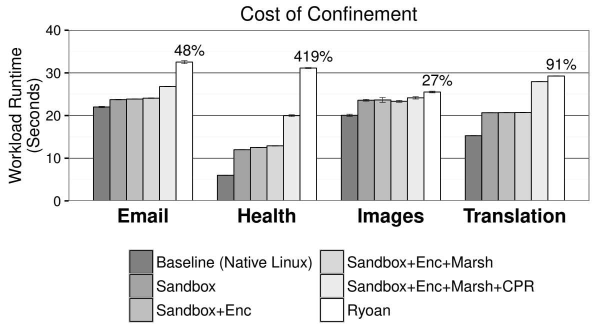


Figure 6: Runtimes of applications with Ryoan sources of overhead broken out. Each bar represents the mean of 5 trials annotated with the 95% confidence interval. Ryoan bars show percent slowdown over native. (Enc: encryption; Marsh: syscall marshaling; CPR: checkpoint restore; Ryoan: Sandbox+Enc+Marsh+CPR+SGX)

		Load Size (MB)	Inited Size (MB)	Init Time (sec)	CPU Time (sec)	CPR Size	Sys. Calls	PF	Intrp
Email	Distribute	18.0	18.1	0.59	1.32	11.6MB	47k	60k	473
	DSPAM	19.6	273.5	11.15	22.10	45.3MB	1.29m	1.81m	6k
	ClamAV	21.1	403.9	24.96	29.17	83.3MB	247k	423k	7k
	Combine	18.0	18.1	0.59	0.11	16KB	12k	2k	77
Health	LoadModel	19.3	19.4	0.58	12.52	28KB	82k	280k	56k
	Classifier	19.3	19.4	0.58	18.23	36KB	1.84m	359k	151k
	Return	18.0	18.1	0.59	6.77	16KB	668K	162k	3k
Images	Distribute	18.0	18.1	0.59	0.42	632KB	2k	2k	36
	Recognize	26.6	27.1	0.63	24.79	83.2MB	88k	174k	6k
	Combine	18.0	18.1	0.59	0.36	2.5MB	14k	3k	129
	Translation	25.3	386.9	2.34	26.65	29.1MB	303k	248k	8k

Email Input	250 emails, 30% with 103KB-12MB attachment
Health Input	20,000 1.4KB Boolean vectors from different users
Images Input	12 images, sizes 17KB-613KB
Trans. Input	30 short paragraphs, sizes 25-300B, 4.1KB total

Table 2: For each workload, report counts of significant events during one execution of each module. Load Size: the size of the loaded module before execution, Inited Size: module size after initialization. Init Time: module initialization time. CPU Time: Processing time of enclave (seconds), CPR size: data copied/zeroed on checkpoint restore, Sys. Calls: system calls, PF: page faults, Intrp: interrupts. "Images: Recognize" reports the maximum of all 4 image recognition enclaves.

mode), augmented with SGX version 2 instructions. We could not use OpenSGX [41], because it lacked 64-bit signals. Our emulator can run a complete software stack including an SGX-aware Linux kernel.

Figure 6 shows a breakdown of the various sources of overheads for Ryoan. The baseline is to run applications built for a native Linux environment and then add sandboxing, encryption, syscall marshaling, checkpoint restore and SGX (where SGX overheads are a mix of emulation and measurements, see the discussion below). Table 2 shows the inputs for each of the workloads, as well as detailed measurements for each module in the DAG and counts of important events (the workloads are explained in Section 7).

Inputs. Workload inputs are designed to be realistic. Email bodies are taken from a spam training set [4]. Email attachments are a set of PDFs randomly attached to 30% of emails (figure taken from a study of corporate email characteristics [12]). Images are a mix of photographs, computer generated patterns, and logos. Gene data was synthesized based on DisGeNet [14]. Translation text comes from the News Commentary dataset [13].

Confinement overhead. In Figure 6, the Sandbox and Sandbox+Enc overheads are necessary for confinement, and across all workloads encryption does not add significant overheads. For Genes, the confinement overhead is high (100%) because it runs a very simple SVM classifier and the actual data processing time is small, which amplifies the effect of Ryoan's data buffering/padding and serves as a worst-case scenario. For Images, the workload involves heavy computation with OpenCV and the confinement overhead is 18%.

Checkpoint restore overhead. The CPR Size column in Table 2 shows the amount of memory copied/zeroed on checkpoint restore. Figure 6 (the difference between the Sandbox+Enc+Marsh and Sandbox+Enc+Marsh+CPR columns) shows that checkpoint restore's impact on performance is significant (55%) for Genes, because it has the lightest per-unit workload (≈1ms) and the relative cost of page fault handling is high; in contrast, its impact on Images is only 3%, which has the heaviest per-unit workload (≈2s).

SGX overhead. Executing code in an SGX-protected enclave imposes several overheads. We simulate SGX hardware overheads by using delays to model the performance of SGX instructions and we flush the TLB on all enclave exits (we could not measure execution on our hardware because it lacks SGX version 2 features (§6.1)). Besides explicit EEXIT instructions, we also model exits due to events like exceptions and interrupts (Table 2). The amount of delay for `EENTER` and `EEXIT` is based on our hardware measurement (3.9μs for each `EENTER`/`EEXIT` pair); in kind, the amount of delay added for each `ERESUME`/Async-Exit pair is based on our hardware measurement (3.14μs).

Version 2 instructions `EACCEPT`, `EMODPE`, `EMODPR` are simpler, so we model their cost at one-tenth of one `EENTER`/`EEXIT` pair. Figure 7 explores the effect of varying this cost on the runtime of our workloads. If the version 2 instruction turn out to be as costly as an `EENTER`/`EEXIT` pair (3.9μs), for instance, the running times of our email, health, images, and translation workloads increase by 25%, 14%, 7%, and 4% respectively. Every checkpoint-related page fault requires one `EMODPE` to extend page permissions. Every page reverted after checkpoint requires one `EMODPE` followed by one `EACCEPT`. Unfortunately version 2 of SGX also imposes extra synchronization (via extended behaviors of `ETRACK`) when modifying enclave page state [56]. We believe the performance effect on these workloads will be negligible, given that our applications only have one thread per

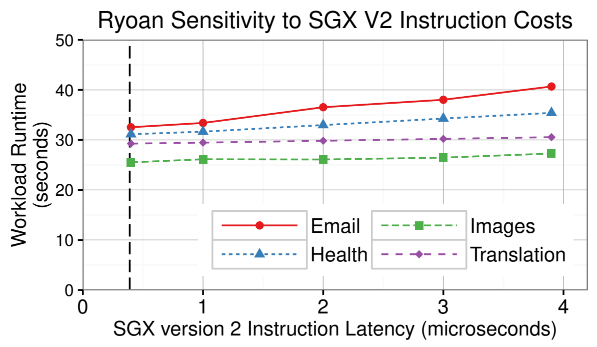


Figure 7: Ryoan application workloads' sensitivity to emulated instruction cost. The dashed vertical line denotes the delay (0.39μs) used to compute the Ryoan bars in figure 6.

enclave. SGX execution also requires syscall marshaling to copy system call arguments and results to and from untrusted memory, but the overhead of marshalling is negligible. All results are shown in Figure 6.

Checkpoint restore vs initialization. Creating an enclave and loading a module takes less than 0.5s for all our cases, but Table 2 shows application-level initialization times are over 20 seconds for DSPAM and ClamAV because they need to load and parse databases. As a result, for this workload it is preferable to use Ryoan's checkpoint-based reset rather than reinitialize the modules for every work unit. Enclave construction imposes further overheads on reinitialization. Even creation of small enclaves (e.g., 298KB) incur a penalty of 30 milliseconds. In comparison, Ryoan's checkpoint-based reset is much more efficient, and the per-unit cost is under 10ms.

9. Related work

Haven [21] allows a trusted program and its library operating system to execute in an SGX enclave that protects them from attack by host software. VC3 [63] secures trusted MapReduce using SGX. MiniBox [48] uses Native Client and a trusted computing module (TPM) to protect an application and the OS from each other. Systems like Overshadow [29], InkTag [40] and Sego [46] use a trusted hypervisor to protect trusted applications from an untrusted operating system, and InkTag/Sego also allow a trusted application to verify untrusted operating system services (e.g., a file system) with help from the hypervisor. These systems are designed to protect trusted applications in an untrusted environment, while Ryoan confines untrusted code that processes sensitive data.

Attempts to use late launch and TPMs (e.g., Flicker [54]) for user assurance suffer from poor usability due to the restricted execution environment (even in their modern incarnations such as Ironclad [39]). Code executing in an enclave can be more complex than what is practical to execute on a TPM. SGX also encrypts data in RAM to keep them secret to an enclave, thus preventing a malicious administrator from monitoring the memory bus to steal secrets, which cannot be guaranteed by TPMs.

Decentralized information flow control (DIFC) allows untrusted applications to access secret data but prevents them from leaking data to unauthorized parties. However, most DIFC systems require that all trusted code is deployed in a centralized platform or administrative domain under a trusted, privileged reference monitor [18, 45, 52, 60, 69, 76]; similar enforcements have also been realized in a browser (COWL [66]) and a mobile device (Maxoid [72]). An exception is DStar [77], which does not have a centralized reference monitor; however, although the user does not need to trust all machines involved in the system, she has to trust the machine that she wants to process her data, which means a correct reference monitor (the OS that supports DIFC) is properly installed on the machine, and that the machine's administrator does not use root privilege to steal secret data. Such trust is not required in Ryoan. Systems that track information flow down to the hardware gate level [67, 68] form a basis for strong information flow guarantees, but such hardware is not available and as designed does not include the privacy and integrity guarantees provided by SGX.

Timing and termination channels are studied in previous work [36, 43] in the context of information flow control. In Ryoan, a module has to terminate for each unit of work, and the processing-time channel can only be used once per unit; different units will not interfere due to module reset.

Homomorphic encryption [25, 38] and order-preserving encryption [22] share similar motivations with Ryoan. They allow untrusted code to perform certain operations directly on encrypted data, in order to protect secrecy. There are also systems built on these primitives [23, 59, 65]. However, these techniques usually suffer from limited application scenarios, weak security guarantees, or significant performance overhead. In comparison, Ryoan's confinement does not require domain-specific knowledge about the applications.

10. Conclusion

Ryoan allows users to safely process their secret data with software they do not trust, executing on a platform they do not control, thereby benefiting users, data processing services, and computational platforms.

11. Acknowledgments

We would like to thank Mark Silberstein, Christopher J. Rossbach, Bennet Yee and Petros Maniatis, the anonymous reviewers and our shepherd Jeff Chase for comments on early revisions of this work. We also thank Shane Williams for background on mail servers. We acknowledge funding from NSF grants CNS-1228843 and CFC-1333594, and a Google research award.

References

[1] 23andMe compares family history and genetic tests for predicting complex disease risk. http://mediacenter.23andme.com/blog/23andme-compares-family-history-and-genetic-tests-for-predicting-complex-disease-risk/. (Accessed: September 2016).

[2] Amazon machine learning. https://aws.amazon.com/machine-learning/. (Accessed: September 2016).

[3] Clarifai. https://www.clarifai.com. (Accessed: September 2016).

[4] CSMINING Group: Spam email datasets. https://csmining.org/index.php/spam-email-datasets-.html. (Accessed: April 2016).

[5] IBM visual recognition service. http://www.ibm.com/smarterplanet/us/en/ibmwatson/developercloud/visual-recognition.html. (Accessed: September 2016).

[6] Implementation and safety of NaCl SFI for x86-64. https://groups.google.com/forum/#!topic/native-client-discuss/C-wXFdR2lf8. (Accessed: September 2016).

[7] Intel(R) Software Guard Extensions for linux* OS, linux-sgx. https://github.com/01org/linux-sgx. (commit:d686fb0).

[8] Intel(R) Software Guard Extensions for linux* OS, linux-sgx-driver. https://github.com/01org/linux-sgx-driver. (commit:0fb8995).

[9] libsodium: A modern and easy-to-use crypto library. https://github.com/jedisct1/libsodium. (Accessed: September 2016).

[10] Moses. http://www.statmt.org/moses/. (Accessed: September 2016).

[11] QEMU: open source processor emulator. http://wiki.qemu.org/Main_Page. (Accessed: September 2016).

[12] The Radicati group, inc: Email statistics report 2009-20013 (summary). http://www.radicati.com/wp/wp-content/uploads/2009/05/email-stats-report-exec-summary.pdf. (Accessed: September 2016).

[13] Shared task: Machine translation. http://www.statmt.org/wmt13/translation-task.html. (Accessed: September 2016).

[14] The DisGeNET Database. http://www.disgenet.org/ds/DisGeNET/files/current/DisGeNET_2016.db.gz. (Accessed: February, 2016).

[15] Intel(R) Software Guard Extensions programming reference, 2014. https://software.intel.com/sites/default/files/managed/48/88/329298-002.pdf.

[16] Intel software guard extensions evaluation SDK users guide: Diffie-Hellman key exchange. https://software.intel.com/sites/products/sgx-sdk-users-guide-windows/Default.htm, 2015. (Accessed: September 2016).

[17] Jason Ansel, Petr Marchenko, Úlfar Erlingsson, Elijah Taylor, Brad Chen, Derek L. Schuff, David Sehr, Cliff L. Biffle, and Bennet Yee. Language-independent sandboxing of just-in-time compilation and self-modifying code. In *PLDI*, 2011.

[18] Owen Arden, Michael D George, Jed Liu, K Vikram, Aslan Askarov, and Andrew C Myers. Sharing mobile code securely with information flow control. In *IEEE S&P*, 2012.

[19] Sergei Arnautov, Bohdan Trach, Franz Gregor, Thomas Knauth, Andre Martin, Christian Priebe, Joshua Lind, Divya Muthukumaran, Daniel O'Keeffe, Mark L. Stillwell, David Goltzsche, Dave Eyers, Rüdiger Kapitza, Peter Pietzuch, and Christof Fetzer. SCONE: Secure linux containers with Intel SGX. In *OSDI*, 2016.

[20] Aslan Askarov, Danfeng Zhang, and Andrew C. Myers. Predictive black-box mitigation of timing channels. In *CCS*, 2010.

[21] Andrew Baumann, Marcus Peinado, and Galen Hunt. Shielding applications from an untrusted cloud with Haven. In *OSDI*, 2014.

[22] Alexandra Boldyreva, Nathan Chenette, Younho Lee, and Adam Oneill. Order-preserving symmetric encryption. In *EuroCrypt*, 2009.

[23] Raphael Bost, Raluca Ada Popa, Stephen Tu, and Shafi Goldwasser. Machine learning classification over encrypted data. In *NDSS*, 2015.

[24] Lèon Bottou. Stochastic gradient SVM. http://leon.bottou.org/projects/sgd#stochastic_gradient_svm. (Accessed: September, 2016).

[25] Zvika Brakerski and Vinod Vaikuntanathan. Fully homomorphic encryption from ring-lwe and security for key dependent messages. In *CRYPTO*. 2011.

[26] Erik Buchanan, Ryan Roemer, Hovav Sacham, and Stefan Savage. When good instructions go bad: Generalizing return-oriented programming to risc. In *CCS*, 2008.

[27] Cristian Cadar, Daniel Dunbar, and Dawson Engler. KLEE: Unassisted and automatic generation of high-coverage tests for complex systems programs. In *OSDI*, 2008.

[28] Stephen Checkoway and Hovav Shacham. Iago attacks: Why the system call API is a bad untrusted RPC interface. In *ASPLOS*, 2013.

[29] Xiaoxin Chen, Tal Garfinkel, E. Christopher Lewis, Pratap Subrahmanyam, Carl A. Waldspurger, Dan Boneh, Jeffery Dwoskin, and Dan R. K. Ports. Overshadow: A virtualization-based approach to retrofitting protection in commodity operating systems. In *ASPLOS*, 2008.

[30] Jim Chow, Ben Pfaff, Tal Garfinkel, Kevin Christopher, and Mendel Rosenblum. Understanding data lifetime via whole system simulation. In *USENIX Security*, 2004.

[31] Bart Coppens, Ingrid Verbauwhede, Koen De Bosschere, and Bjorn De Sutter. Practical mitigations for timing-based

side-channel attacks on modern x86 processors. In *IEEE S&P*, 2009.

[32] Victor Costan and Srinivas Devadas. *Intel SGX Explained*. 2016.

[33] Victor Costan, Ilia Lebedev, and Srinivas Devadas. Sanctum: Minimal hardware extensions for strong software isolation. In *USENIX Security*, 2016.

[34] Solar Designer. ”return-to-libc” attack. Bugtraq, 1997.

[35] Andrew Ferraiuolo, Yao Wang, Danfeng Zhang, Andrew C. Myers, and G. Edward Suh. Lattice priority scheduling: Low-overhead timing-channel protection for a shared memory controller. In *HPCA*, 2016.

[36] Bryan Ford. Plugging side-channel leaks with timing information flow control. In *HotCloud*, 2010.

[37] Hubertus Franke, Rusty Russell, and Matthew Kirkwood. Fuss, futexes and furwocks: Fast userlevel locking in linux. In *Ottawa Linux Symposium*, 2002.

[38] Craig Gentry. *A fully homomorphic encryption scheme*. PhD thesis, Stanford University, 2009. `https://crypto.stanford.edu/craig`.

[39] Chris Hawblitzel, Jon Howell, Jacob R. Lorch, Arjun Narayan, Bryan Parno, Danfeng Zhang, and Brian Zill. Ironclad apps: End-to-end security via automated full-system verification. In *OSDI*, 2014.

[40] Owen S. Hofmann, Sangman Kim, Alan M. Dunn, Michael Z. Lee, and Emmett Witchel. InkTag: Secure applications on an untrusted operating system. In *ASPLOS*, 2013.

[41] Prerit Jain, Soham Desai, Seongmin Kim, Ming-Wei Shih, JaeHyuk Lee, Changho Choi, Youjung Shin, Taesoo Kim, Brent Byunghoon Kang, and Dongsu Han. OpenSGX: An Open Platform for SGX Research. In *NDSS*, San Diego, CA, February 2016.

[42] Min Gyung Kang, Stephen McCamant, Pongsin Poosankam, and Dawn Song. DTA++: Dynamic taint analysis with targeted control-flow propagation. In *NDSS*, 2011.

[43] Vineeth Kashyap, Ben Wiedermann, and Ben Hardekopf. Timing-and termination-sensitive secure information flow: Exploring a new approach. In *IEEE S&P*, 2011.

[44] Taesoo Kim, Marcus Peinado, and Gloria Mainar-Ruiz. STEALTHMEM: system-level protection against cache-based side channel attacks in the cloud. In *USENIX Security*, 2012.

[45] Maxwell Krohn, Alexander Yip, Micah Brodsky, Natan Cliffer, M. Frans, Kaashoek Eddie, and Kohler Robert Morris. Information flow control for standard OS abstractions. In *SOSP*, 2007.

[46] Youngjin Kwon, Alan Dunn, Michael Lee, Owen Hofmann, Yuanzhong Xu, and Emmett Witchel. Sego: Pervasive trusted metadata for efficiently verified untrusted system services. In *ASPLOS*, 2016.

[47] Butler W. Lampson. A note on the confinement problem. *CACM*, 16(10), October 1973.

[48] Yanlin Li, Jonathan McCune, James Newsome, Adrian Perrig, Brandon Baker, and Will Drewry. MiniBox: A two-way sandbox for x86 native code. In *USENIX ATC*, 2014.

[49] Anyi Liu, Jim Chen, and Harry Wechsler. Real-time covert timing channel detection in networked virtual environments. In *International Conference on Digital Forensics*, 2013.

[50] Chang Liu, Michael Hicks, Austin Harris, Mohit Tiwari, Martin Maas, and Elaine Shi. GhostRider: A hardware-software system for memory timerace oblivious computation. In *ASPLOS*, 2015.

[51] Fangfei Liu, Hao Wu, and Ruby B. Lee. Can randomized mapping secure instruction caches from side-channel attacks? In *HASP*, 2015.

[52] Jed Liu, Michael D George, Krishnaprasad Vikram, Xin Qi, Lucas Waye, and Andrew C Myers. Fabric: A platform for secure distributed computation and storage. In *SOSP*, 2009.

[53] Jay Mahadeokar and Gerry Pesavento. Open sourcing a deep learning solution for detecting NSFW images. `https://yahooeng.tumblr.com/post/151148689421/open-sourcing-a-deep-learning-solution-for`. (Accessed: September 2016).

[54] Jonathan M. McCune, Bryan Parno, Adrian Perrig, Michael K. Reiter, and Hiroshi Isozaki. Flicker: An execution infrastructure for TCB minimization. In *EuroSys*, April 2008.

[55] David A. McGrew and John Viega. The Galois/Counter mode of operation (GCM). `http://csrc.nist.gov/groups/ST/toolkit/BCM/documents/proposedmodes/gcm/gcm-revised-spec.pdf`, 2005. (Accessed: September 2016).

[56] Frank McKeen, Ilya Alexandrovich, Ittai Anati, Dror Caspi, Simon Johnson, Rebekah Leslie-Hurd, and Carlos Rozas. Intel software guard extensions (intel sgx) support for dynamic memory management inside an enclave. In *HASP*, 2016.

[57] Andrew C. Myers and Barbara Liskov. A decentralized model for information flow control. In *SOSP*, 1997.

[58] Olga Ohrimenko, Felix Schuster, Cdric Fournet, Sebastian Nowozin Aastha Mehta, Kapil Vaswani, and Manuel Costa. Oblivious multi-party machine learning on trusted processors. In *USENIX Security*, 2016.

[59] Raluca Ada Popa, Catherine Redfield, Nickolai Zeldovich, and Hari Balakrishnan. CryptDB: protecting confidentiality with encrypted query processing. In *SOSP*, 2011.

[60] Donald E. Porter, Michael D. Bond, Indrajit Roy, Kathryn S. McKinley, and Emmett Witchel. Practical fine-grained information flow control using laminar. *TOPLAS*, 37(1), 2014.

[61] Ashay Rane, Calvin Lin, and Mohit Tiwari. Raccoon: Closing digital side-channels through obfuscated execution. In *USENIX Security*, 2015.

[62] Ling Ren, Christopher W. Fletcher, Albert Kwon, Emil Stefanov, Elaine Shi, Marten van Dijk, and Srinivas De-

vadas. Constants Count: Practical improvements to oblivious RAM. In *USENIX Security*, 2015.

[63] Felix Schuster, Manuel Costa, Cedric Fournet, Christos Gkantsidis, Marcus Peinado, Gloria Mainar-Ruiz, and Mark Russinovich. VC3: Trustworthy data analytics in the cloud using SGX. In *IEEE S&P*, 2015.

[64] David Sehr, Robert Muth, Cliff Biffle, Victor Khimenko, Egor Pasko, Karl Schimpf, Bennet Yee, and Brad Chen. Adapting software fault isolation to contemporary cpu architectures. In *USENIX Security*, 2010.

[65] Justine Sherry, Chang Lan, Raluca Ada Popa, and Sylvia Ratnasamy. Blindbox: Deep packet inspection over encrypted traffic. In *SIGCOMM*, 2015.

[66] Deian Stefan, Edward Z Yang, Petr Marchenko, Alejandro Russo, Dave Herman, Brad Karp, and David Mazieres. Protecting users by confining JavaScript with COWL. In *OSDI*, 2014.

[67] Mohit Tiwari, Xun Li, Hassan M. G. Wassel, Frederic T. Chong, and Timothy Sherwood. Execution leases: A hardware-supported mechanism for enforcing strong non-interference. In *MICRO*, 2009.

[68] Mohit Tiwari, Jason K. Oberg, Xun Li, Jonathan Valamehr, Timothy Levin, Ben Hardekopf, Ryan Kastner, Frederic T. Chong, and Timothy Sherwood. Crafting a usable microkernel, processor, and i/o system with strict and provable information flow security. In *ISCA*, 2011.

[69] Steve Vandebogart, Petros Efstathopoulos, Eddie Kohler, Maxwell Krohn, Cliff Frey, David Ziegler, Frans Kaashoek, Robert Morris, and David Mazières. Labels and event processes in the asbestos operating system. *TOCS*, 25(4), December 2007.

[70] Zhenyu Wu and Zhang Xu. Whispers in the hyper-space: High-bandwidth and reliable covert channel attacks inside the cloud. *TON*, 23(2), April 2015.

[71] Yuanzhong Xu, Weidong Cui, and Marcus Peinado. Controlled-channel attacks: Deterministic side channels for untrusted operating systems. In *IEEE S&P*, 2015.

[72] Yuanzhong Xu and Emmett Witchel. Maxoid: Transparently confining mobile applications with custom views of state. In *EuroSys*, 2015.

[73] Yunjing Xu, Michael Bailey, Farnam Jahanian, Kaustubh Joshi, Matti Hiltunen, and Richard Schlilchting. An exploration of l2 cache covert channels in virtualized environments. In *CCSW*, 2011.

[74] Bennet Yee, David Sehr, Gregory Dardyk, J. Bradley Chen, Robert Muth, Tavis Ormandy, Shiki Okasaka, Neha Narula, and Nicholas Fullagar. Native Client: A sandbox for portable, untrusted x86 native code. In *IEEE S&P*, 2009.

[75] Tatu Ylonen and Chris Lonvick. RFC 5246: The Transport Layer Security (TLS) Protocol: Version 1.2. `https://tools.ietf.org/html/rfc5246`, August 2008. (Accessed: September 2016).

[76] Nickolai Zeldovich, Silas Boyd-wickizer, Eddie Kohler, and David Mazires. Making information flow explicit in histar. In *OSDI*, pages 263–278. USENIX Association, 2006.

[77] Nickolai Zeldovich, Silas Boyd-Wickizer, and David Mazieres. Securing distributed systems with information flow control. In *NSDI*, 2008.

[78] Chao Zhang, Tao Wei, Zhaofeng Chen, Lei Duan, Laszlo Szekeres, Stephen McCamant, Dawn Song, and Wei Zou. Practical control flow integrity and randomization for binary executables. In *IEEE S&P*, 2013.

[79] Danfeng Zhang, Aslan Askarov, and Andrew C. Myers. Predictive mitigation of timing channels in interactive systems. In *CCS*, 2011.

[80] Danfeng Zhang, Aslan Askarov, and Andrew C. Myers. Language-based control and mitigation of timing channels. In *PLDI*, 2012.

[81] Y. Zhang, A. Juels, M. K. Reiter, and T. Ristenpart. Cross-VM side channels and their use to extract private keys. In *CCS*, 2012.

Unobservable communication over fully untrusted infrastructure

Sebastian Angel
UT Austin and NYU

Srinath Setty
Microsoft Research

Abstract

Keeping communication private has become increasingly important in an era of mass surveillance and state-sponsored attacks. While hiding the contents of a conversation has well-known solutions, hiding the associated metadata (participants, duration, etc.) remains a challenge, especially if one cannot trust ISPs or proxy servers. This paper describes a communication system called Pung that provably hides all content and metadata while withstanding global adversaries. Pung is a key-value store where clients deposit and retrieve messages without anyone—including Pung's servers—learning of the existence of a conversation. Pung is based on private information retrieval, which we make more practical for our setting with new techniques. These include a private multi-retrieval scheme, an application of the power of two choices, and batch codes. These extensions allow Pung to handle $10^3\times$ more users than prior systems with a similar threat model.

1 Introduction

Can two or more users exchange messages over a public network without anyone else learning that they communicated? And can this be done in a practical manner without trusting any other entities (e.g., other users, ISPs, proxy servers)? This paper answers these questions affirmatively with a communication system that provides strong privacy guarantees, even against active, global adversaries.

While the questions we consider are decades old [32], there is a renewed interest motivated by an increase in service providers disclosing their users' information without consent [7, 13, 16, 83, 90, 109], as well as questionable mass surveillance practices [15, 27, 50, 62, 63] that defy existing privacy laws and long-held beliefs [44, 116, 128, 129, 133]. In response, companies have mobilized to deploy end-to-end encryption solutions to safeguard the privacy of users' communications [1–3, 5, 61]. While end-to-end encryption protects the content of the messages exchanged, it does not hide their existence nor other metadata (e.g., identity of participants, duration), which can be just as sensitive [38, 88, 117, 120].

Fortunately, the threat of metadata leakage has not been lost on academics and practitioners; there is a vast literature on preventing such disclosures [22, 25, 31–33, 40–43, 47, 51, 75, 79, 81, 82, 92, 93, 98, 113, 114, 119, 121, 130, 136]. While these works make great strides toward providing strong guarantees and supporting many users, we find that most require trusting one or more entities in the communication infrastructure (e.g., proxy servers, ISPs, large coalitions of users) to achieve their goals. In many contexts, such assumptions can be sensible. However, deployment considerations such as "where to find a trusted entity or an incorruptible consortium to run the system" are often left unspecified and are arguably hard to answer. Furthermore, there is enough precedent to think that private communication is a setting where trustworthiness can be subverted by financial and political interests [13, 39, 83, 90, 118]. There are proposals that do not require trusting the communication infrastructure [26, 33, 42, 60, 66, 131], but they have been primarily theoretical since the resulting systems support only dozens of concurrent users.

This tension between trust and performance drives our work. Our view is that private communication can be achieved with reasonable performance, even in the presence of strong adversaries. To substantiate this position we build Pung, a system that provably hides all metadata associated with users' conversations—even against adversaries who control all the communication infrastructure (ISPs, cloud providers, etc.) and arbitrary coalitions of users. We find that a 4-server deployment of Pung supports 135K messages/minute with 32K active users: $10^5\times$ more messages and $10^3\times$ more users than any prior system that withstands a similar adversary (§7.3). When we extend this comparison to systems under weaker threat models we find that Pung is promising but is not yet a replacement: Pung handles $85\times$ fewer users (§7.2).

To support tens of thousands of users at modest costs, Pung addresses two challenges. The first is architectural: devising a way for users to send and receive messages without a trusted proxy. Our proposal is simple, and consists of combining untrusted servers and powerful cryptography through a synthesis of known ideas (§3). The second, and more salient aspect of Pung is reducing the costs of its cryptographic machinery. Our contributions here include algorithms that amortize expensive operations when users send and receive multiple messages (§4).

In more detail, Pung is an untrusted key-value store that exposes private deposit and retrieval procedures to users. Pung's deposit procedure is based on the ability of communicating users to agree on a shared *label* (or "key" in the key-value store) under which to store a message (§3.1). Pung's retrieval procedure builds on a powerful—but expensive—cryptographic primitive: private information retrieval (PIR) [36]. PIR allows clients to fetch items from a server without revealing to the server which items were fetched. While PIR has been used in other private communication systems [79, 92, 119], its interface is not a good fit for Pung: PIR requires clients to know the exact index of the items they wish to retrieve in a data structure stored at the server. In Pung, this data structure is continuously modified, and clients know only a label (§3.1).

To improve the performance of PIR, Pung targets applications where users retrieve multiple messages: email, group chats, bug reporting, and sensor data collection (§8). Pung then introduces a private multi-retrieval scheme that departs from most prior approaches (e.g., [18, 64, 67, 86]) in two ways. First, instead of modifying the design or implementation of PIR, Pung encodes the underlying data structures; these techniques are independent of the PIR scheme used (§4.1). Second, Pung leverages an inherent property of private communication systems: to resist traffic analysis they operate in rounds in which a bounded number of messages is sent and received by each user (§3.1). Users who wish to send or receives messages past this limit must wait several rounds to do so. Pung exploits this restriction with a multi-retrieval scheme that is probabilistically—rather than perfectly—complete: in a few cases clients can only retrieve a fraction of the items they wish to retrieve, but they can try again later. This results in a more efficient scheme than all prior PIR schemes that support multi-retrievals (§4.2).

To integrate PIR with Pung, we adapt an existing oblivious search technique [35] that allows clients to retrieve messages with labels (§3.3), and extend it to work on the encoded data structures that Pung uses for multi-retrieval (§4.4). Pung also introduces several other features. First, Pung supports group communication. Second, Pung provides a service that allows users to privately derive a shared secret to bootstrap their conversations, provided they know each other's public keys (§6). Last, messages in Pung are long-lived and can be retrieved at a later time by clients who participate infrequently (§8).

Nonetheless, our work has several limitations. While we reduce costs compared to prior approaches, these costs—especially network costs—remain high (§7.4). Furthermore, many of our techniques are only beneficial when clients retrieve multiple messages. Like all past private communication systems, Pung does not hide the fact that users are part of the system (it only hides if and with whom they are communicating); users are also required to participate even when they have nothing to send or retrieve. Pung does not provide liveness guarantees (censorship resistance) in the face of malicious servers or ISPs. This is fundamental since an ISP could simply refuse to route network packets. Lastly, Pung does not currently support an efficient dialing protocol to enable clients to "cold-call" one another (§5). Despite these limitations, we believe that Pung takes an important step toward enabling untrusted private communication.

2 Goals and threat model

In this section we discuss our goals and assumptions, and the general ecosystem that Pung targets.

2.1 Private communication over the Internet

Our objective is to develop a messaging system that allows two or more users to communicate over the Internet (or any other public network) while hiding the content of all messages exchanged in addition to the metadata of the exchange. The types of metadata that we wish to keep hidden from anyone—except from the users directly involved—include the start and end time of a conversation, the number of messages exchanged, the identity of the participants, etc. Some of this information is difficult to keep private since many existing services rely on it for their proper functioning. For instance, ISPs need to know destinations to route packets, email and chat service operators—who would in principle deploy and manage Pung—need to know the messages that make up a conversation, etc. Consequently, Pung must balance the requirements of existing services and infrastructure with the preservation of the following security goals:

Message integrity and privacy. The content of a message must be intelligible only to its intended recipient. Furthermore, no one should be able to tamper with a message while it is in transit without the recipient being able to detect alterations. Specifically, we target the strongest cryptographic properties that capture these goals, namely integrity of ciphertexts under chosen plaintext attacks (INT-CTXT) [21, 70], and indistinguishability under adaptive chosen ciphertext attacks (IND-CCA2) [99, 110].

Metadata privacy. An adversary must not be able to determine if (or when) a user sent or received a message. Furthermore, an adversary must not be able to link a message exchange with the users that participated in that exchange. Specifically, we target the privacy notion of *relationship unobservability* as defined by Pfitzmann and Hansen [105]. Informally, relationship unobservability states that an adversary does not learn useful information from observing (or actively interfering with) all network traffic, provided that the sender and the recipient are not compromised. In the case of such compromise, relationship unobservability offers little value: the sender could trivially disclose that it is sending a message and to whom; a receiver could similarly leak the sender's identity.

The above restriction is consistent with our setting of two-way communication. However, as we note in Section 9, relationship unobservability is not a panacea. For instance, it is not on its own sufficient to protect whistleblowers who wish to remain anonymous from everyone—including all recipients. We give a formal definition of metadata privacy and provide proofs of security for all of our techniques in our extended report [12, Appendix A].

2.2 Security assumptions

Pung achieves the security properties above under the following set of assumptions.

Cryptographic assumptions. Pung requires an authenticated encryption scheme (e.g., [21, 53]) to meet our goals of message integrity and privacy. Pung also relies on a computational private information retrieval (CPIR) scheme (e.g., [10, 58, 73]) and a pseudorandom function (e.g., [19, 20]) for ensuring metadata privacy (§3.1–§3.3).

Trust assumptions. Pung assumes that users who wish to communicate know their peer's public key (or can exchange a secret through and out-of-band channel). Pung provides privacy guarantees only to pairs (or groups) of users who communicate through Pung while following the prescribed protocol. However, these guarantees are not predicated on the behavior of any other user in the system, or the communication channel between users. In particular, Pung's guarantees hold even if *all* of the infrastructure that Pung uses (servers, ISPs, DNS, etc.) is compromised and operates arbitrarily.

Liveness assumption. Pung assumes that services used by clients to communicate with each other do not deny service. That is, we expect ISPs to carry traffic, DNS to provide name resolution, and servers to process requests. While this assumption is not needed for Pung to meet our security goals (§2.1), it is essential for Pung to be usable.

3 Design and architecture

Pung adopts a client-server architecture in which third-party servers mediate the exchange of messages between users. Figure 1 depicts this architecture. From the perspective of end users, a Pung cluster acts as a storage service. This parallels services like Gmail or Outlook that store messages on behalf of users.

Users exchange messages via a Pung client application that deposits the messages into *mailboxes* in the Pung cluster. These mailboxes are addressed by a *label* that is known to both the sender and the recipient. Recipients can access a message sent to them by retrieving the contents of a mailbox from the Pung cluster using an appropriate label. Pung's "mailbox" architecture is borrowed from prior systems [25, 40, 75, 79, 119, 130]. A key difference is which entities run the storage nodes, the kinds of processing that these nodes do, and the mechanisms for storing and retrieving messages. We discuss each of these components in the following sections, but we first highlight how this architecture fits within our target ecosystem.

Pung's mailbox architecture forces all messages sent and retrieved to go through entities like ISPs and the Pung cluster. These services rely on (or can easily infer) the types of metadata that we wish to hide, since they process all network traffic. Consequently, protecting metadata without harming the functioning of these services requires that the rate at which clients send and receive network packets be disentangled from the rate at which they send and retrieve messages in Pung. This requirement is key to preventing many types of traffic analysis attacks [68, 96, 112]. Unfortunately, it results in an unavoidable inefficiency: clients must send and retrieve messages at an independent (e.g., constant, Poisson) rate, *even* when the user is idle. This requires that clients queue excess requests and add cover traffic or *chaff* [115] (fake requests that are indistinguishable from real ones).

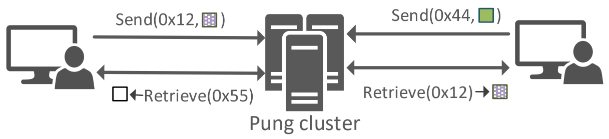


FIGURE 1—Client applications issue send and retrieve requests to the Pung cluster at a given rate, introducing fake requests whenever the user is idle (or issues fewer requests than the rate).

We now discuss how mailbox labels are derived, and how clients can use them to send and retrieve messages.

3.1 Mailbox labels and discretized rounds

The Pung protocol proceeds in discretized rounds or time epochs. Round duration is configurable and depends on the use case. The Pung cluster acts as a point of synchronization for clients and dictates when a new round starts. While this allows the Pung cluster to force clients out of sync, doing so results in a denial of service but does not violate our goals (§2.1). During each round, client applications issue exactly one send and one retrieve. This ensures that clients issue requests at a constant rate (§3). In Section 4 we relax this model and let clients issue multiple send and retrieve requests per round, enabling several applications (§8) and achieving lower (amortized) costs (§7.3). Finally, Section 5 discusses how clients can manage existing connections, and how they can agree on a round on which to start a new conversation.

Deriving mailbox labels. The Pung cluster is effectively a key-value store that treats mailbox labels as keys, and (encrypted) messages as values. This means that users' communication depends on their ability to agree on a label under which to store and retrieve messages. This label should be unique (to avoid multiple pairs of users overwriting each other's messages), and it must also be independent of the users communicating (otherwise an adversary could link a label to a conversation). Pung achieves both of these properties through a combination of shared secrets and a pseudorandom function (PRF).

Recall from Section 2.2 that we assume that users who wish to communicate have access to each other's public key (e.g., RSA key), or have exchanged a secret through an out-of-band channel. In Section 6 we present a directory service that allows users to derive a shared secret directly from public keys. Consequently, the rest of this section assumes that users have a shared secret which acts as a master key. This master key is used to derive two additional keys, k_L and k_E, with a key derivation scheme [76]. The derived keys are used for mailbox label generation and message encryption, respectively. We also assume that users have a unique identifier, *uid*, within each pair of communicating users. For example, if Alice and Bob wish to communicate with each other, Alice could be "0" and Bob could be "1". This information need not be private, so users could choose any identification scheme including using their names or public keys.

Each user can derive the corresponding labels for the current round r, $label_S(r)$ and $label_R(r)$, by invoking the pseudorandom function (PRF) keyed with k_L:

$$label_S(r) = \mathrm{PRF}_{k_L}(r \,||\, uid_{peer})$$
$$label_R(r) = \mathrm{PRF}_{k_L}(r \,||\, uid_{own})$$

where r is a fixed-width integer and $||$ is the concatenation operator when r and *uid* are treated as binary strings. Note that labels need not be symmetric: a user can send a message to Alice and retrieve one from Bob in the same round. In such cases, the labels would be generated using different keys and *uid*s. If a user is idle and has nothing to send or retrieve, it generates random mailbox labels.

3.2 Sending messages in Pung

Sending a message in Pung consists of deriving the recipient's mailbox label ($label_S$), and encrypting the message with an authenticated encryption scheme (§2.2) using key k_E. The client then sends the resulting ciphertext, $c = AE(k_E, m)$, along with the mailbox label, to the Pung cluster as a $(label_S, c)$-tuple. Idle users send a tuple that consists of a random label and an encryption of a random message instead. We assume that all messages are the same size or that padding is applied.

3.3 Retrieving messages from the Pung cluster

Observe that if the Pung cluster were to broadcast to all users the $(label, c)$-tuples received during a round, users could iterate through the list locally and find the tuple with the label that is of interest to them (or determine that it is not present). Intuitively, this operation would not leak any information about which label (if any) was of interest to a retriever, and would not allow the adversary to determine with whom a user is communicating (or if the user is idle). Of course, broadcasting all tuples would incur prohibitive network costs. Fortunately, retrieving an item from an untrusted server without revealing *which* item was retrieved is the problem addressed by private information retrieval (PIR) [36]. PIR protocols trade off computation at the server to achieve lower network costs than the above broadcast scheme. We summarize PIR next, since it is the basis of message retrieval in Pung.

Private information retrieval (PIR). We focus on *computational* PIR (CPIR) schemes [10, 28, 30, 57, 58, 73, 77, 135] that hide users' access patterns under cryptographic hardness assumptions.[1] At a high level, a CPIR scheme operates over a collection DB of n items held by a server, and consists of three procedures: QUERY, ANSWER, DECODE. The $\textsc{Query}(idx, n)$ procedure is run by the client; it outputs a query q that encodes the index, idx, in DB of the desired element. The $\textsc{Answer}(q, DB)$ procedure is run by the server; it returns an encrypted response a that contains the element in DB at the index encoded in q. This step requires the server to perform cryptographic operations over *all* elements in DB. The $\textsc{Decode}(a)$ procedure is run by the client; it decrypts a to recover the desired element in DB. Below we describe a simple CPIR scheme based on an additively homomorphic cryptosystem.[2]

The client first generates a *query vector* q of length n by calling the $\textsc{Query}(idx, n)$ procedure. Every entry in q is a different encryption of 0, except for the entry at position idx which is an encryption of 1. The client sends this query to the server, who executes $\textsc{Answer}(q, DB)$ to produce a ciphertext c that encrypts the element in DB at position idx. To do this, the server creates a vector x by interpreting every entry $e_i \in DB$ as an integer, and computing the product of e_i and the ciphertext q_i. This can be accomplished through repeated additions of q_i by leveraging the additive homomorphic property of the cryptosystem: $x_i = \prod_1^{e_i} q_i$. The server then adds up every entry in x to obtain a. This procedure works because the vector x consists of $n-1$ ciphertexts that encode 0, and one ciphertext that encodes e_{idx}. Adding all of them results in an encryption of e_{idx}, without the server learning which index was requested. Lastly, the client runs $\textsc{Decode}(a)$ to decrypt a and get the desired element.

All of the CPIR schemes to which we refer (and on which we rely) are more efficient than the above straw man, but they have a similar flavor. Crucially, they enjoy communication costs sublinear in n (i.e., they are cheaper than transferring the entire collection). Furthermore, some CPIR schemes (e.g., [10]) have low enough computational costs that their processing latency is actually lower than transferring the entire collection over today's networks (this was believed to be an unlikely scenario [125]).

Retrieving messages. Since PIR allows clients to privately retrieve an item from the server at some index, one possibility is to use labels as indices: clients can retrieve a message from $label_R(r)$ with $q = \textsc{Query}(label_R(r))$. However, the size of the collection would need to match

[1] IT-PIR schemes [36, 48, 59] are an efficient alternative to CPIR but rely on multiple servers, at least one of which must be correct. This conflicts with our goals and threat model (§2).

[2] An additively homomorphic cryptosystem supports an operation "$\cdot$" that can be used on ciphertexts to produce a new ciphertext encoding the sum of their plaintexts. That is, $Enc(x) \cdot Enc(y) = Enc(x + y)$.

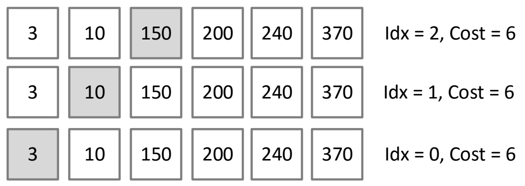


FIGURE 2—A client wishing to retrieve an item with label "3" from a server holding a sorted list of 6 items would need to perform three rounds of probing. During each probe, the client guesses an index, uses PIR to retrieve the (*label*, *c*)-tuple at that index, and refines the guess accordingly. "Cost" indicates the number of items processed by the server in each probe.

the range of the labels (§3.1), which is 256 bits in our implementation (§6). This would require Pung servers to materialize and operate over a collection of 2^{256} items!

Instead, we can arrange for Pung servers to insert all (*label*, *c*)-tuples sent by clients in some search data structure (e.g., sorted list, search tree) and present them as a collection *DB* of size *n* (where *n* is the total number of nodes in the data structure). This enables clients and Pung servers to perform PIR directly on *DB*, but there is a problem: clients know from which label they wish to retrieve, but they do not know the mapping between labels and the index of the desired tuple in the data structure representing *DB*, or if the tuple even exists. This can easily be addressed by having clients obtain this label-to-index mapping from the Pung cluster. However, when the collection is large (n>100K), clients can use a search scheme to reduce network costs. We discuss this below.

The key idea is that clients can find their desired element in *DB* via an "oblivious" search. Figure 2 depicts an example of this search when *DB* is stored as a sorted list. In this case, the client performs $\log(n)$ probes to locate its desired element (or determine that it is not present). Even if the client gets lucky and finds its element early, it must continue until the end to preserve privacy; the remaining probes can just use any indices. Since each probe is a PIR query to the entire collection *DB*, the server must process *n* elements each time; the time complexity of this search is therefore $\Theta(n\log(n))$. However, this scheme has a lot of redundancy: the server processes each item $\log(n)$ times. Chor et al. [35] show that one can eliminate this "double counting" overhead by using data structures that can be (logically) split into independent chunks while retaining the search capability. We elaborate on this idea below in the context of the specific construction that Pung uses.

BST retrieval. We choose to use a complete[3] binary search tree (BST) as our underlying data structure for several reasons. First, a complete BST is balanced, enabling search in $\mathcal{O}(\log(n))$ probes. Second, for any dataset there is a unique complete BST, so the Pung cluster need not

[3]A k-ary tree is complete if all of its levels (except possibly the last) are full, and the last level is filled from left to right.

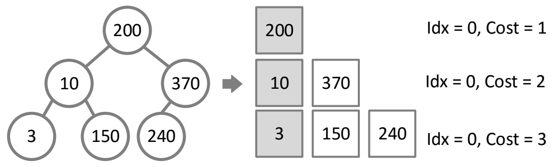


FIGURE 3—The Pung cluster can store (*label*, *c*)-tuples in a complete BST, allowing clients to treat each level as an independent collection. Clients can issue a PIR query for the top level, and can recursively derive the index of lower levels using BST semantics. This figure depicts the search for label "3".

```
function BST-RETRIEVAL($L^\star, n$)
    $h \leftarrow \lfloor log_2(n) \rfloor$                // last level of the BST
    $c^\star \leftarrow \perp$          // target ciphertext ($\perp$ means not yet found)
    $idx \leftarrow 0$                       // index of the current level
    $len \leftarrow 1$                      // length of the current level
    $len^h \leftarrow n - (2^h - 1)$              // length of the last level

    for $i = 0$ to $h$ do
  // use PIR to get element at position $idx$ from collection at level $i$
        $q \leftarrow$ QUERY($idx, len$)
        $a \leftarrow$ send $i$ and $q$ to server and get answer
        $(L, c) \leftarrow$ DECODE($a$)

        if $c^\star == \perp$ then
            if $L^\star < L$ then                  // access left child next
                $idx \leftarrow 2 \cdot idx$
            else if $L^\star > L$ then            // access right child next
                $idx \leftarrow 2 \cdot idx + 1$
            else              // $L^\star == L$, found target ciphertext
                $c^\star \leftarrow c$

        $len \leftarrow 2^{i+1}$ or $len^h$                // length of the next level

        if $idx \geq len$ or $c^\star \neq \perp$ then
            $idx \leftarrow$ random index between 0 and $len - 1$
    return $c^\star$
```

FIGURE 4—Client procedure for retrieving an encrypted message $c^\star$ from a mailbox with label $L^\star$. The server holds a collection of *n* (*label*, *c*)-tuples in a complete binary search tree.

communicate the structure to clients (aside from *n*). Last, since every level of a complete BST is full (except for possibly the last) and every node contains an actual data item, there is no need for padding or auxiliary elements; it can be represented as a contiguous array without overhead.

Based on this, we set up the Pung cluster to store the collection of (*label*, *c*)-tuples in a complete BST, and have clients treat all the nodes at the same depth in the tree (i.e., on the same level) as a (logically) separate collection. As depicted in Figure 3, clients can then process each of the $\log(n)$ collections sequentially from top to bottom, deriving the index of the next level from the semantics of the BST. The pseudocode for this procedure is listed in Figure 4. Since each collection (and therefore each element) is accessed exactly once, there is no overhead due to double counting. Indeed, the time complexity of this BST-based retrieval scheme is $\Theta(n)$, which is the same as if the clients had known the index in the first place.

Compared to performing PIR over a known index, clients do incur a $\log(n)\times$ higher network cost due to retrieving a tuple at every level. As an optimization, clients could fetch (non-privately) all of the tuples of the first few levels, saving both bandwidth and CPU. This is because CPIR queries and answers are typically much larger than the elements in the collection; when the collection is small, it is more efficient to download all elements (i.e., naive PIR) than to use a CPIR scheme (§7.4).

The above sending and retrieval procedures are sufficient to build a version of Pung that meets all of our security goals (§2.1): it enables users to communicate with each other privately, hiding the content and preserving the integrity of messages, without leaking any metadata associated with a conversation. Furthermore, none of the security guarantees depend on the correctness of the Pung cluster. For instance, if the Pung cluster modifies the ciphertext associated with any tuple, clients can detect this due to the integrity guarantees of the authenticated encryption scheme. If the server drops tuples or stores them in a data structure that is not a complete BST, clients will be unable to find the tuple of interest to them (a denial of service), but the integrity of the content and the privacy of the communication is preserved. The drawback with the above scheme is its costs: the server has to process the entire collection for each client request. Additionally, for applications where clients wish to retrieve more than one message in a round (§8), costs scale linearly with the number of messages retrieved. The next section describes ways to significantly amortize costs for regimes in which clients retrieve multiple messages simultaneously.

4 Reducing costs via multi-retrievals

This section describes how to reduce the CPU costs of the Pung cluster when clients retrieve multiple messages.

4.1 Prior approaches to multi-retrieval

One approach to retrieving k items from the server is to run the protocol in Section 3.3 k times, but this results in costs that are linear in k. An alternative is to create new PIR schemes that support a batch of k retrievals with sublinear costs. Groth et al. [64] achieve significant improvements with this approach, but their focus is reducing network costs—the resulting CPU overheads are prohibitive in our context. Another approach is to modify the implementation, rather than the design, of existing PIR schemes. In particular, as we discuss in Section 3.3, the query of many PIR schemes is a vector of encrypted entries. The server can aggregate the queries submitted by (potentially different) users into batches of size k, and construct a matrix. This enables the server to leverage fast matrix multiplication algorithms (e.g., Strassen's algorithm [126]) to evaluate PIR's ANSWER procedure. Several works have shown that this yields modest benefits [18, 67, 86].

In Pung, we take a different approach—inspired by *batch codes* [69]—from the schemes above: instead of modifying the design or the implementation of a particular PIR protocol, we focus solely on changing the representation of the underlying data.[4] We discuss batch codes in detail in Section 4.4 since we use them as a final refinement to our scheme. At a high level, they enable the server to encode a collection into smaller subcollections, in such a way that clients can retrieve any k items by querying each subcollection at most once. Below we highlight several reasons for designing a new mechanism rather than directly applying batch codes.

Challenges and opportunities. First, many batch codes suffer from a major drawback: the number of elements that a client downloads increases rapidly with k. This means that for small k (3 or 4), network costs are within a small factor of retrieving items one by one; but they quickly rise to untenable levels with larger k. Second, batch codes' *perfect completeness* guarantee (i.e., that clients can retrieve any k items) is too conservative for our setting. In particular, Pung does not require that clients can *always* retrieve all k messages during a given round: since messages in Pung are long-lived (§6), clients can retry the next round. This behavior is actually inevitable in systems resistant to traffic analysis, such as Pung: recall that clients send and retrieve messages at some rate; any client who receives messages in excess of this rate must wait at least two rounds. Below we describe an alternative that works well for larger k, but is probabilistic. That is, a client can sometimes only retrieve a subset of the k messages that it wished to retrieve in a given round.

4.2 Probabilistic private multi-retrieval

We now introduce a new probabilistic *multi-retrieval* scheme. A multi-retrieval scheme allows a server to efficiently process multiple retrievals from the *same* client by amortizing costs. Our proposal is more efficient than prior approaches, especially for larger values of k (> 4).

At the core of multi-retrieval is the observation that as long as every item in the server's collection is processed at least once, the underlying PIR protocol will ensure that the server does not learn which tuples were retrieved. As we discuss in Section 3.3, one can take a collection and structure it as a tree, allowing each level to be treated independently. This results in clients retrieving $\log(n)$ tuples, while the server processes each element just once; incurring the same CPU costs as a single retrieval. The reason that BST-RETRIEVAL (Fig. 4) is not technically a multi-retrieval scheme is that clients have no control over which tuples are fetched (they are forced to follow BST semantics), and consequently the procedure can only

[4]Using PIR as a black box means that other optimizations (e.g., fast matrix multiplication) benefit Pung as well.

output a single message. We now show a way to divide the collection into smaller subcollections while still allowing clients some control over which items to fetch.

Server setup. The server initially performs a static partitioning of the label space (e.g., 2^{256}) into B buckets (we set B to the maximum number of messages that users retrieve in a round, i.e. k). Each bucket holds all $(label, c)$-tuples whose labels fall into its partition. At the end of the send phase, the server takes all the $(label, c)$-tuples sent by clients and distributes them across the B buckets based on their label. Small buckets store tuples in an arbitrary order, while larger buckets store tuples in an array that represents a complete BST (§3.3). The latter enables clients to use BST-RETRIEVAL (Fig. 4), which saves network resources. Finally, the server sends clients the number of items in each bucket, and awaits retrieval requests.

Client lookup. A client can retrieve multiple messages simultaneously by treating each bucket as an independent collection and retrieving one $(label, c)$-tuple from each bucket. This is done by calling an appropriate retrieval procedure on each bucket with a label that falls within the bucket's range and the size of the bucket: for BST-encoded buckets, the client uses BST-RETRIEVAL; for other buckets, the client requests the label-to-index mapping, and retrieves a $(label, c)$-tuple by directly sending the output of PIR's QUERY procedure to the server. If a client does not wish to retrieve a tuple from a particular bucket, it performs the retrieval using a random label. Note that since BST-RETRIEVAL (or PIR's QUERY) is executed on each bucket independently, the server's CPU cost is still the same as if the client had requested a single tuple from the entire collection (as was the case in §3.3).

In the best case, since there are as many buckets as user queries ($B = k$), clients can retrieve all of their desired messages at once. However, this scenario presupposes that all tuples that the client wishes to retrieve have labels that fall in different buckets. But what if a client wished to retrieve ρ tuples ($1 < \rho \leq k$) from the same bucket?

Unfortunately this cannot be done privately as it would require the client to interact with the same bucket ρ times, leaking information about the requested labels. Instead, the entire protocol must be rerun ρ times, allowing the user to retrieve one message from the contested bucket on each run. There is one caveat: the number of times that the protocol is rerun during a round must not depend on the user's choice of labels; this too would leak information. Instead, the number of reruns must be set a priori.

But how common is it for clients to want to retrieve multiple tuples from the same bucket? This is a standard balls-and-bins scenario, since the client's labels are generated from a pseudorandom function, and the buckets' range is statically and independently partitioned. We can thus bound the number of tuples that fall in any bucket by $\rho \leq \frac{3\ln(k)}{\ln(\ln(k))}$ [95, Lemma 5.1]; this bound fails to hold with probability $\leq \frac{1}{k}$. Unfortunately, this is a fairly large number (9–11, for $k \leq 512$), especially since we require rerunning the entire protocol ρ times to guarantee that clients can retrieve k messages with high probability.

Below we describe how Pung reduces the bound on ρ exponentially by reaping the load balancing benefits of giving clients multiple choices to retrieve tuples [94].

4.3 Fewer reruns with the power of two choices

Azar et al. [14] show that in a k balls and k bins scenario, if each ball maps to d random bins ($d > 1$), and balls are placed in the bin least full, the highest load in any bin is bounded by $\frac{ln(ln(k))}{ln(d)} + \Theta(1)$ with high probability.

We observe that if clients had multiple buckets from which to retrieve a message, we could apply this result to decrease the bound on ρ, and consequently the number of reruns that clients must perform during multi-retrieval (§4.2). However, this kind of load balancing is typically applied from the *producer's* perspective (e.g., choosing which server to issue a request, or on which queue to place a packet); in our case, we are interested in enabling the *consumer* (i.e., the recipient of a message).

This raises the following question: how can we enable a client to be able to retrieve a message under two labels? We propose a seemingly bad idea: have senders derive *two* labels for each message, and have the server store messages under both labels. This of course doubles the already large number of messages in the system (n). Considering that all PIR costs scale linearly with n, and the BST retrieval scheme (§3.3) adds a multiplicative $\log(n)$ factor to network costs, this is a cause for concern. However, the exponential decrease in the number of reruns that clients will have to perform (i.e., ρ), far outweighs the costs associated with doubling all messages. Ultimately, this simple approach results in significant savings.

We implement the above scheme by extending Pung's send and retrieve procedures (§3.2). Recall that clients derive two keys from their shared secret, and use one of them (with a PRF) to generate a label under which to store a message. Under the modified protocol, clients derive a third key that they use in combination with a second PRF to generate the extra label.[5] Clients can then send (L_1, L_2, c) to the server, which then stores c under two different $(label, c)$-tuples. During retrieval, clients generate both labels for each message they wish to retrieve (§3.3) and follow the lookup scheme (§4.2) using the label that leads to fewer bucket collisions. Note that collisions are defined with respect to a client's other labels. They are independent of the actions of other clients or the server; they are therefore a notion local to each client.

[5]Clients need to ensure that both labels do not map to the same bucket. This can be done by using a counter as a nonce to the PRF, incrementing it until both labels map to different buckets.

4.4 Probabilistic multi-retrieval with batch codes

The above bucket-based scheme makes progress toward lowering CPU and network costs, but still requires the protocol to be rerun ρ times. In this section we further refine the scheme by composing it with batch codes, discussed next, to achieve a hybrid scheme that has lower CPU costs than either mechanism, fewer round trips than the bucket-based scheme, and lower network costs than applying existing batch codes in isolation.

Batch codes. A (n, N, k, m)-batch code [69] takes as input a collection of n items and the number of desired retrievals k ($k > 1$), and outputs N items ($n < N < n \cdot k$) distributed across m subcollections ($m > 2$) that have a useful load-balancing property: any k items from the original collection can be retrieved by querying each of the subcollections at most once. In our context, this means that a Pung server that encodes n $(label, c)$-tuples with a batch code can process k simultaneous queries from the same client, while only paying the processing cost required to answer one query to a collection of N tuples.

We now give an example of a $(n, \frac{3}{2}n, 2, 3)$-batch code scheme that supports $k = 2$ retrievals. A collection DB of n items, is split into 3 subcollections db_1, db_2, db_3, such that db_1 has the first half of the items, db_2 has the second half of the items, and db_3 has $db_1 \oplus db_2$ (where $\oplus$ is the element-wise XOR operator). A single PIR query to each subcollection is thus sufficient to privately retrieve any two items from DB (we provide details later in this section). Furthermore, the CPU cost of answering all three queries (one for each subcollection) is the same as that of processing one PIR query over a collection of $N = \frac{3}{2}n$ items. Therefore, this scheme is 25% cheaper than running PIR twice on DB to retrieve 2 items (since that would require processing $2n$ items).

Subcube batch codes [69] are a generalization of this scheme and allow clients to retrieve any k items at once by recursively performing the above encoding (e.g., to support $k = 4$, one encodes each of db_1, db_2, db_3 to obtain a total of $m = 9$ subcollections). Consequently, large values of k significantly amortize the CPU cost of retrieving k items. A disadvantage is that clients always have to retrieve an element from each of the m subcollections, where $m = 3^{log(k)}$ in the above scheme. This is acceptable for small k, but for large k the network overheads are enormous: for $k = 128$, clients retrieve $17\times$ more elements than running 128 instances of the scheme in Section 3.3.[6]

On the other hand, our probabilistic bucket-based scheme allows clients to retrieve k messages at once with lower CPU and network overhead, but requires ρ reruns of the protocol (ρ is roughly 3–4 with our refinement in §4.3). The rationale behind rerunning the protocol is that clients might need to retrieve up to ρ items from the same bucket. Observe that retrieving a few items (e.g., $k \approx$ 2–4) is a strength of subcube batch codes. It therefore makes sense to hybridize the two techniques. However, subcube batch codes are not compatible with BST-based retrieval (which reduces network costs for large buckets as discussed in §3.3). We address this with the following technique, which might be of independent interest.

BST retrieval with subcube batch codes. We now adapt BST-RETRIEVAL (Fig. 4) to work on encoded collections. We focus on the $(n, \frac{3}{2}n, 2, 3)$-subcube batch code described earlier but our approach generalizes.

Server setup. The server starts with a collection of n $(label, c)$-tuples, which it sorts based on labels. Analogous to the batch code scheme described earlier, the server splits the collection into two halves, and stores them as two complete BSTs, b_1 and b_2. Finally, the server creates a third binary tree, b_3, from b_1 and b_2 as follows: for every level i and index j, $b_3(i,j) = b_1(i,j) \oplus b_2(i,j)$. The server then indicates to clients the collection size (n) and the lowest label in b_2, L_{mid}; tuples with labels lower than L_{mid}, if they exist, would be found in b_1.

Client lookup. A client wishing to retrieve two tuples labeled L_1 and L_2 can do so as follows. Assume without loss of generality that $L_1 < L_2$. There are two cases:

- If $L_1 < L_{mid}$ and $L_2 \geq L_{mid}$: the client calls BST-RETRIEVAL(L, $\frac{n}{2}$) on each tree independently, passing L_1 for b_1, L_2 for b_2, and a random label for b_3.
- If $L_1 < L_{mid}$ and $L_2 < L_{mid}$, the client calls BST-RETRIEVAL(L_1, $\frac{n}{2}$) on b_1, and performs a *joint tree traversal* on b_2 and b_3 to retrieve L_2 (the case where both $L_1 \geq L_{mid}$ and $L_2 \geq L_{mid}$ is symmetric and simply requires exchanging the role of b_1 and b_2).

Joint tree traversal. Since b_3 is not a BST (i.e., the order of its elements does not respect BST semantics), it cannot be used directly for search. However, it can be jointly traversed with the help of another tree. We describe this for the case where $L_1 < L_{mid}$ and $L_2 < L_{mid}$. A client starts by retrieving the tuples at level 0 and index 0 for both b_2 and b_3 in parallel. This is equivalent to lines 10–12 in Figure 4 (during the first iteration of the loop). The result of the two separate calls (one for each tree) to the DECODE procedure in line 12 is the pair of tuples t_2 and t_3. While the label of t_3 is unintelligible (since it is encoded) and the label of t_2 is irrelevant to the client's search, they can be combined to compute $(L, c) = t_1 = t_2 \oplus t_3$, which is the corresponding tuple in b_1. This yields a way to jointly traverse the trees: the client can compare L_2 to L and choose whether to go left or right on both b_2 and b_3 for the next level. If $L_2 = L$, the client can save c (as this is the desired ciphertext), and continue with random indices for the remaining levels. The above steps are analogous to lines 14–25 in Figure 4 when one replaces $L^\star$ with L_2.

[6]Other batch codes exist [69, 104, 111, 123], but their concrete costs are significantly higher than those of subcube batch codes in all our cases.

A hybrid scheme. As before, the server partitions the label space into B buckets. For each bucket b, the server encodes all the corresponding tuples with a (n_b, N_b, ρ, m)-subcube batch code. Here, n_b is the initial number of tuples in b, ρ is the number of reruns required after deriving two labels per tuple (§4.3), N_b is the total number of tuples in b after encoding, and m is the number of subcollections per bucket ($m = 3^{log(\rho)}$). If n_b is large enough, the server uses the BST-aware batch code presented above so clients can benefit from the lower network cost of BST-based retrieval. The upshot is that combining batch codes with probabilistic multi-retrieval lets clients retrieve up to ρ tuples from each bucket, without rerunning the protocol.

5 Operational challenges

A key challenge in any communication system is managing user connections. In particular, how do clients determine when and for how long to communicate? In Pung, the answer depends on the type of pre-existing relationship that users have: *symmetric*, where users already know each other and have already derived a shared secret (§3.1), and *asymmetric*, where one user wishes to "cold call" another for the first time. We now describe both cases.

Managing symmetric connections. Client applications of users who already know each other can exchange *control messages* through Pung. Control messages have a special structure that client applications can recognize and automatically act upon, so they are transparent to actual users. Control messages are sent over Pung like any other message—so they too are private—and include statements like "`END`" to indicate that a conversation is over, or "`START [round]`" to indicate the round when a conversation should start. These messages are sent periodically (e.g., every 20 rounds), but can also be sent during an active communication in response to events (e.g., `END` is sent when the application is placed in the background or when the user stops typing for a few minutes).

The frequency of control messages is initially configured the first time that two users communicate with each other, but it can be adjusted dynamically with the "`FREQ [rounds]`" control statement. Higher frequency leads to smoother operation (e.g., client applications can agree on a round to start a conversation faster), but like any other message, they count toward the send and retrieve rate limit chosen by the user (§3.1). Pung's multi-retrieval optimizations (§4) make sending and receiving control messages more efficient, and enable clients to fetch control messages from several known peers at once.

Initiating asymmetric connections. The exchange of control messages described above presupposes an established relationship between clients. But how does Pung bootstrap this interaction in the first place? One option is for clients to use control messages to introduce their peers to others. A more realistic alternative is for clients to use a *dialing* protocol, as proposed by Vuvuzela [130] and Alpenhorn [80]. In a dialing protocol, clients send *invitations* (messages stating the desire of a user to start a conversation, and information about a round on which to do so) to mailboxes with labels derived from users' email addresses [80] or public keys [130]. Clients can then periodically check their corresponding mailboxes for invitations, without leaking metadata in the process.

Unfortunately, Pung does not currently support an efficient dialing protocol. We attempted to adapt Vuvuzela's dialing scheme, but due to Pung's threat model and architecture, we found that it degenerates into each client having to download the invitations sent by all users. The precise issue is that Pung does not provide sender anonymity [105]. Incidentally, all existing systems that provide sender anonymity without trusted infrastructure are fully peer-to-peer and broadcast messages to everyone [33, 42, 60, 66, 131]. This makes dialing gratuitous since all users already know each other (i.e., relationships are symmetric), and they actively communicate with everyone in every round. Designing an efficient dialing scheme under our setting (§2)—or proving that it cannot exist—remains an open question.

6 Implementation

We implement Pung in 5,800 lines of Rust and C++ bindings. We express the server-side computation of Pung in Naiad's timely dataflow model [97], and use the Timely Dataflow library [89] written in Rust, to create, run, and coordinate dataflow workers. Each worker processes send and retrieve requests issued by clients, encodes the tuple collections, and invokes the PIR procedures exposed by XPIR [11]. Finally, we derive keys from secrets with HKDF [76], generate labels with HMAC-SHA256, and encrypt messages with ChaCha20-Poly1305. All of these operations are supported by the Rust-Crypto library [8].

Additional features. Our prototype supports:
• *Long-lived messages.* The Pung cluster maintains a sliding window of messages, regardless of the number of rounds over which they were sent. This allows users to retrieve messages sent to them during past rounds. This requires dataflow workers to mix new and existing messages, garbage collect the messages that outlive the sliding window, and reconstruct buckets and BSTs.
• *Group communication.* Pung provides privacy to groups if all users in the group follow the protocol. Suppose a group G has derived a shared key k_L, then: (1) user $i \in G$ can send its message to G under label $\text{PRF}_{k_L}(r \mid\mid uid_i)$ during round r; (2) users in G can simultaneously retrieve all messages sent in round r using a multi-retrieval query with labels $\text{PRF}_{k_L}(r \mid\mid uid_j)$ for all $j \in G$.
• *Directory service.* If users know each others' public

keys pk_i (e.g., RSA keys), they can derive a shared secret through a standard Diffie-Hellman key exchange [49] via Pung. User i can send the tuple $(\text{PRF}_0(pk_i),\{pub_i,\sigma_i\})$ to the server, where pub_i corresponds to i's public Diffie-Hellman parameters $(g, p, g^a \bmod p)$, and σ_i is a signature of pub_i under i's private key. Notice that the tuple's label depends only on pk_i; anyone with access to pk_i can derive the label and retrieve the tuple. Clients can retrieve each other's public components (pub_i), verify their authenticity, and derive the shared secret independently. Clients send these tuples to Pung servers when they first register, or via a special message that flags them so they are not garbage collected by dataflow workers. Pung stores these tuples in the same collection as other messages, so their access is kept private. If the tuples are larger than regular messages, they are split into chunks; clients can retrieve these chunks over several rounds or with multi-retrieval.

Compressing explicit label mappings. Recall that for large collections BST retrieval incurs less network costs than explicitly downloading the label-to-index mappings and performing PIR with a known index (§3.3). We now describe how to delay the *breakeven point* (i.e., the collection size at which BST retrieval is better than explicitly downloading labels) by using a *Bloom filter* [24]. A Bloom filter is a probabilistic data structure that encodes a compressed representation of a set, and is widely used to reduce network costs in many settings, including private communication [80, 108] (although our use case is different). It exposes a *check* procedure that allows anyone to check whether some element is in the set (false positives are possible and occur with small probability).

In our implementation, the Pung server adds to a Bloom filter the element $index||label$ for each tuple in the collection, and sends it to clients. Clients can then find the index of their desired label $L^\star$ by testing for set membership locally while varying the index until a match is found: $check(0||L^\star), \ldots, check(n-1||L^\star)$. While standard Bloom filters require computing a large number of hash functions for each add and check operation, there exist constructions that require only two [74]. Thus, with little computation, clients can locally derive their desired index while saving network resources. For larger collections, retrieval via BST (Fig. 4) remains more efficient.

7 Experimental evaluation

Our evaluation answers four main questions. First, what is the cost of the cryptographic primitives used in Pung (§7.1)? Second, what is the concrete performance of Pung, and how does it compare to prior systems (§7.2)? Third, what are the benefits of multi-retrieval (§7.3)? Last, what are the costs that Pung imposes on clients (§7.4)?

Setup and metrics. We deploy Pung's server logic on timely dataflow workers running on Microsoft Azure H16 instances (16-core Intel Xeon E5-2667 with 112 GB RAM) with Ubuntu 16.04. Our performance metrics are throughput (in messages/minute) and end-to-end latency (in seconds). Note that all entities run on the same data center, so our results do not capture the effects of wide area networking. In all cases we report the mean over 10 trials; standard deviations are less than 10% of the means.

We run clients and dataflow workers in a closed loop and let round duration be as low as possible: a new round starts as soon as all current requests are fulfilled. To keep the number of messages constant across rounds, we configure Pung's garbage collection window to be the number of messages sent in one round (§6).

Baselines. We compare Pung to two prior systems: Dissent [42] and Vuvuzela [130]. They represent the state-of-the-art in private communication under the anytrust[7] (Vuvuzela) and no-trust (Dissent) models. We want to emphasize that our comparison to Dissent is not apples-to-apples: Dissent achieves an additional privacy property—sender anonymity (§2, §9)—that Pung does not provide. However, we are not aware of a system with the same guarantees as Pung under our threat model.

7.1 Microbenchmarks

To understand the costs of Pung we start with a series of microbenchmarks. The network and CPU costs of many of Pung's operations depend on the size of the collection (n = # of tuples) held by the Pung cluster and the size of each $(label, c)$-tuple. We report the results for several collection sizes, and tuple sizes (288 bytes, 1 KB). We choose these tuple sizes to match our baselines: Vuvuzela clients exchange 256-byte encrypted messages (Pung's 32-byte labels account for the difference), while Dissent targets larger messages ($\geq$ 1 KB). The costs of PIR operations depend on two parameters: aggregation (α) and dimension (d) [10]. They control the number of ciphertexts that make up a PIR query and answer (higher α and d lead to smaller queries but larger answers). For each collection and tuple size, Pung dynamically chooses the parameters that minimize total network costs.

Figure 5 tabulates our results. We find that client-side operations incur little CPU costs aside from generating a PIR query. This operation is performed once by clients when retrieving a message, or several times (on smaller collections) when traversing a BST (§3.3). The network and CPU cost of generating and sending a PIR query depend on the number and the size of the ciphertexts that make up the query; for the PIR parameters that Pung uses (last two rows of Figure 5), these costs are sublinear in the size of the collection (i.e., $\sqrt{n}$). We discuss more about client-to-server network costs in Section 7.4.

[7]The anytrust model [137] states that out of a set of servers one is assumed to be correct; clients need not know which is the correct one.

	# tuples in Pung cluster (n)		
	2,048	8,192	32,768
client-side CPU costs			
Key derivation	6.05 μs	6.05 μs	6.05 μs
Label generation	1.60 μs	1.60 μs	1.60 μs
Message encryption	1.56 μs	1.56 μs	1.56 μs
Message decryption	1.37 μs	1.37 μs	1.37 μs
Bloom filter lookup	0.15 ms	0.47 ms	2.02 ms
PIR query (288 B tuples)	0.86 ms	1.91 ms	3.35 ms
PIR query (1 KB tuples)	1.68 ms	3.36 ms	5.02 ms
PIR decode (288 B tuples)	0.62 ms	0.69 ms	0.70 ms
PIR decode (1 KB tuples)	0.68 ms	0.69 ms	1.35 ms
server-side CPU costs			
PIR setup (288 B tuples)	4.52 ms	16.01 ms	68.73 ms
PIR setup (1 KB tuples)	15.64 ms	63.86 ms	255.38 ms
PIR answer (288 B tuples)	6.05 ms	14.91 ms	36.81 ms
PIR answer (1 KB tuples)	14.72 ms	37.87 ms	143.38 ms
network costs			
PIR query (288 B tuples)	256 KB	512 KB	1024 KB
PIR query (1 KB tuples)	512 KB	1,024 KB	1,536 KB
PIR answer (288 B tuples)	432 KB	464 KB	464 KB
PIR answer (1 KB tuples)	464 KB	464 KB	912 KB
PIR parameters (α, d) [10]			
288 B tuples	$(32, 2)$	$(32, 2)$	$(32, 2)$
1 KB tuples	$(8, 2)$	$(8, 2)$	$(16, 2)$

FIGURE 5—Microbenchmarks for Pung's operations under varying collection sizes (n), and tuple sizes (288 bytes and 1 KB).

Unlike clients' CPU costs, the server's costs are significant. One of the most expensive operation is the one-time setup of a PIR collection. In Pung, this procedure needs to be performed once at the beginning of every round following the send phase (§3.3). The other major source of overhead is answering PIR queries. In general, this cost scales linearly with n, though fixed costs make processing several small collections ($n < 8$K) relatively more expensive than processing a single large one. We return to this point in Section 7.3 when we discuss the theoretical versus actual benefits of our optimizations.

7.2 End-to-end performance of single retrievals

We focus on two end-to-end metrics: latency observed by a client and throughput achieved by Pung servers. Here we test the version of Pung that we describe in Section 3 without any of the multi-retrieval optimizations (§4).

Latency. To measure the end-to-end latency perceived by clients in Pung, we set up a single dataflow worker that is under-utilized and that can immediately handle a user's request. We then have a single client send its message and perform a retrieval. To experiment with large collection sizes we populate the server with up to 1 million 288-byte tuples. We experiment with three different methods that clients can use to retrieve their desired tuples from the server. The first has the client explicitly download all the label-to-index mappings prior to retrieval, look up the index of the corresponding label locally, and perform PIR with this index. The second downloads a Bloom filter that succinctly encodes the label-to-index mappings (§6), and performs the same steps as above. The last performs the BST retrieval procedure listed in Figure 4.

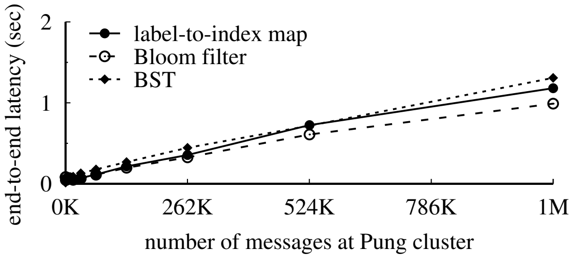


FIGURE 6—The end-to-end latency of sending and retrieving one message when the Pung cluster is under-utilized is up to 1.3 seconds (when the server stores 1 million tuples).

Figure 6 depicts the results. As we expect from our microbenchmarks, the client latency grows linearly with the number of messages at the server. Also, our low-latency network allows us to confirm that the server-side CPU costs associated with BST retrieval are negligibly higher than explicitly fetching the label-to-index mapping. However, in wide area networks we expect to see added latency due to $\log(n)$ round trips. The Bloom filter's checks (§6) also incur little CPU overhead, and its size is up to 10.4× smaller than the associated label-to-index mapping. Finally, note that our prototype performs request-level—rather than data-level—parallelism, so these latencies could be reduced further by having dataflow workers process fractions of a request. However, current latencies are already comparable to those achieved by Vuvuzela, where even a two-client scenario requires 20-second rounds due to the addition and serial processing of cover traffic.

Throughput. To measure Pung's peak throughput, we run experiments where clients send and retrieve a 256-byte message per round, for a total of 10 rounds. We then vary the number of clients (n) and measure the number of messages processed per minute. We distribute 64 timely dataflow workers across 4 VMs to run Pung's server-side computation. Since we cannot run tens of thousands of clients in our infrastructure, we employ a combination of real and simulated clients. We configure 512 real clients across 8 VMs (4 clients per core). We then have each client send a single message and instruct dataflow workers to make up the difference by injecting the remaining messages ($n-512$) at the end of the send phase, simulating additional clients. Finally, during the retrieve phase, each real client fetches a message from a random mailbox.

We also run both baselines in our cluster, with 256-byte messages. Since Dissent is a peer-to-peer system and does not use servers, we spread out its peers across our VMs. We run only its shuffle protocol as that is more efficient than full Dissent for small fixed-sized messages [42, §3].

For Vuvuzela, we set up a 3-server chain in addition to

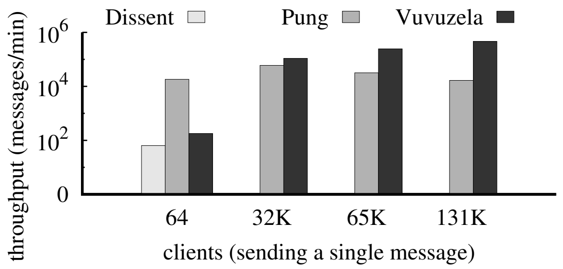


FIGURE 7—Pung can handle significantly more messages and clients than Dissent but its throughput at 131K clients is 27.8× lower than Vuvuzela's. We do not report Dissent's throughput past 64 users (see text for details).

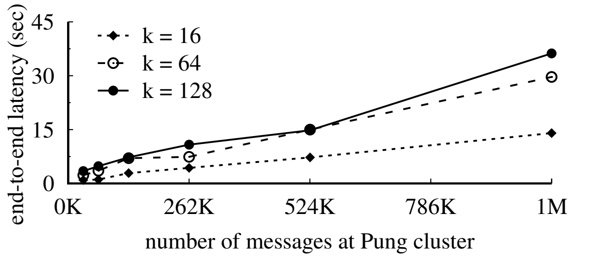


FIGURE 8—The end-to-end latency of sending one message and retrieving k using Pung's multi-retrieval. It takes 36.2 seconds with $k = 128$ and 1M tuples. This is 4.6× faster than retrieving 128 messages using Pung's single-retrieval (Fig. 6).

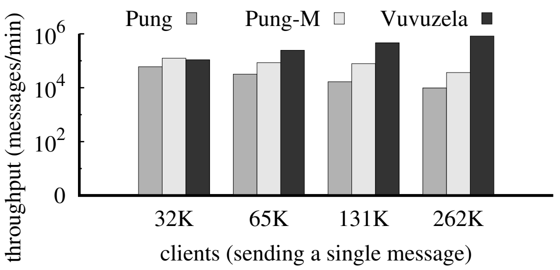


FIGURE 9—Pung's multi-retrieval optimizations increase its throughput by up to 5.2×. Pung-M represents a version of Pung where clients retrieve k=64 messages simultaneously using our hybrid scheme (§4.4). At 262K clients, Vuvuzela handles 84.9× and 22.6× more messages than Pung and Pung-M, respectively.

the entry server that proxies client requests, which mirrors the arrangement evaluated by its authors [130, §7]. A caveat is that our VMs have fewer CPU cores. We also use the same parameters that characterize the distribution from which Vuvuzela servers draw noise ($\mu = 300{,}000$ and $b = 13{,}800$). We run 512 Vuvuzela clients and modify the entry server [9] to make up for the remaining messages (similar to how Pung's dataflow workers inject messages).

Figure 7 depicts our results for 64, 32K, 65K, and 131K clients. We show Dissent's throughput only with 64 clients because at higher peer counts it is less than one message per minute with the prototype we use [6].

Pung and Vuvuzela achieve relatively low throughput—far below their capacity—at very low client counts. This is due to lack of work, since only 64 clients are sending and retrieving messages in a given round. As a result, Pung workers sit idle most of the time, while Vuvuzela servers continue to generate and process significant cover traffic, delaying the start of the next round. However, at higher (and more realistic) client counts, there is enough work to make long rounds a non-issue for Vuvuzela. Indeed, Vuvuzela's throughput is 27.8× higher than Pung at 131K clients, and this gap grows even larger with more clients.

7.3 What are the benefits of multi-retrieval?

We now discuss how our techniques (§4) impact the performance of Pung in terms of latency and throughput. In both cases, we run the same experiments described in Section 7.2, but configure clients to use the hybrid scheme (§4.4) to retrieve multiple messages at once.

Latency. As with the single retrieval case, client latency grows linearly with the number of messages at the server. This is depicted in Figure 8. However, with one million tuples, the multi-retrieval latency is 1.5×, 2.8×, and 4.6× lower than running the single retrieval protocol (§7.2) k times when retrieving $k = 16$, 64, and 128 messages respectively. Note that in this experiment we have a single dataflow worker respond to all of the client's queries (recall that there is a query for each subcollection). However, this is an embarrassingly parallel task since subcollections are independent; different workers could be assigned to each of them. Given enough workers, it is possible to drive down the end-to-end latency of processing all k requests to the level of processing a single request.

Throughput. We depict the throughput benefits of having clients retrieve a batch of $k = 64$ messages in Figure 9. We find that Pung's hybrid scheme offers a throughput boost of up to 5.2× over single retrieval. Based on our cost model (available in our extended report [12, Appendix B]), the maximum gain that we can expect from using our hybrid scheme over retrieving messages one by one is 14.2× for $k = 64$. This large disagreement (over 2×) with our experimental results comes from two main sources. First, our end-to-end throughput measures not only message retrieval but also Pung's send phase—including the expensive PIR setup step (§7.1) and the encoding of buckets using batch codes (§4.4)—which lowers our potential gains. Second, as we discuss in Section 7.1, smaller collections are disproportionately more expensive to serve than larger ones, owing to fixed costs.

Nevertheless, Pung's multi-retrieval throughput is high enough (5.9× lower than Vuvuzela's at 131K clients) that it can accommodate thousands of users and tens of thousands of messages with sub-minute latencies. This performance is sufficient to support many existing applications (§8). We also experiment with values of k ranging from 4 to 128, and find gains between 1.52×–11×.

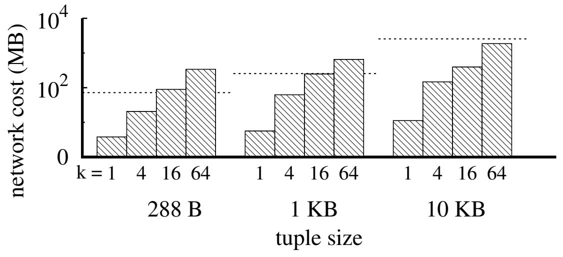


FIGURE 10—Pung's network costs (upload and download) for $n = 262$K with varying k and tuple sizes. The dashed line represents the cost of naively downloading the entire collection, which provides information-theoretic privacy. Pung's single retrieval is cheaper than naively downloading the entire collection. For $k>1$, Pung performs better than naive download only when messages are large, or when k is moderate (see text for details).

7.4 What costs does Pung impose on clients?

Pung's clients have to participate in every round to ensure unobservability (§3.1). Clients thus pay fixed CPU and network costs regardless of their actions. Our microbenchmarks (§7.1) show that many of these costs are small. Indeed, clients incur tens of milliseconds of CPU time per round for the experiments in Sections 7.2 and 7.3.

Network costs. To better understand the network costs incurred by clients, we run a set of experiments in which we vary the collection sizes (n), the number of messages retrieved by a client (k), and the size of tuples in the collection. Figure 10 summarizes the results for $n = 262$K tuples with varying k and the size of tuples.

We find that for single retrievals ($k = 1$), clients incur 3.8–11 MB of network costs for sending and receiving a message, depending on the tuple size. This cost is 3–4 orders of magnitude higher than retrieving the tuple from the server non-privately. However, compared to downloading the entire collection (which would also meet our privacy goals), it is 19× lower for 288 byte tuples, 45× lower for 1 KB tuples, and 230× lower for 10 KB tuples.

For $k>1$, we find that clients incur 4.5–36 MB per message depending on k and tuple size. Perhaps surprisingly, we find that under certain regimes (e.g., small tuple sizes, high k), it is beneficial for clients to simply download the entire collection instead of using Pung's multi-retrieval. The reason is that clients have to retrieve tuples from many subcollections—the number of which depends on k (§4.4)—by sending PIR queries and receiving PIR answers (several ciphertexts). With the PIR construction that we employ (i.e., XPIR [10]), ciphertexts are rather large (128 Kbits), so these overheads are more than the size of the collection for smaller tuple sizes and large k. While we can use a different cryptosystem with smaller ciphertexts (e.g., Paillier [101]) to reduce network costs by orders of magnitude, it incurs much higher server-side CPU costs [10]. We are investigating ways to resolve this conflict between network and CPU costs.

Admittedly, this is the primary limitation of Pung's current design. However, there are certain regimes in which Pung's multi-retrieval outperforms downloading the entire collection: larger messages (e.g., $\geq$1 KB), or medium k (e.g., $\leq$64). For example, with $k = 16$ and 10 KB messages, the total network cost is 7× lower than downloading the entire collection. Finally, while these costs may be considered modest for well-connected devices, they remain high for many settings (e.g., mobile devices).

8 Applicable scenarios

Section 7.3 demonstrates that Pung's optimizations can substantially increase its throughput, but they incur additional network resources and require clients to retrieve many messages at once. We now discuss applications that can benefit from Pung's privacy guarantees as well as its multi-retrieval—high network costs remain an issue.

First, participants in a dark pool (a private stock exchange) could hide their orders using Pung, preventing market speculation and predatory tactics by high-frequency traders [85, 103]. Second, email, group chats, and collaboration tools such as Slack [4] are all a natural fit for Pung: they use larger messages (>1 KB), and require (or benefit from) multi-retrieval.

Finally, several applications with many-to-one communication can use Pung. For instance, health/embedded devices can send diagnostic information to medical providers using Pung, preserving the privacy of the communication. Similarly, Pung enables private collection of data from sensors (e.g., Internet of things), or corporate software (e.g., bug reports). While these devices have limited resources (e.g., power, bandwidth) they can still use Pung, since they can choose (a priori) how often to participate (e.g., every 5 rounds). They can then leverage Pung's multi-retrieval to "catch up" by simultaneously retrieving all messages sent to them during the last 5 rounds. Of course, if a client rarely participates, its messages might be garbage collected before it can catch up (§6).

9 Related work

This section discusses related systems, and their comparison to Pung. (Danezis et al. [45] provide a more thorough discussion of many of these systems.)

Mix networks. The earliest private messaging systems employ *mix networks* [22, 23, 31, 32, 47, 65, 72, 81, 82]: they rely on a set of servers (called mixes) to shuffle messages before delivering them to recipients. This shuffling is often accompanied by encryption, batching, and chaffing (the addition of dummy traffic) to prevent traffic analysis. Since all operations are relatively lightweight, these systems enjoy lower latency and higher throughput than many other works in the literature—including Pung. However, malicious mixes can replay, duplicate, or drop messages, violating these systems' guarantees via known attacks [84, 87, 100, 106, 107, 112, 122, 134]. Indeed, Kesdogan et al. [71] show that many of these attacks are fundamental. Consequently, systems like Aqua [82] and Herd [81] sidestep these attacks by targeting scenarios where particular mixes with critical roles are trusted. The use of such trusted mixes contradict our goals (§2).

There are works with a decentralized architecture: peer-to-peer mix networks [114, 138] and peer-to-peer routing [17, 37, 46, 55, 56, 98, 113, 121]. These systems have high network costs, and rely on a threshold of peers being correct. Furthermore, they are susceptible to strong adversaries [54, 91, 124] and Sybil attacks [52]. Salsa [98] combats these issues by making an additional assumption: fewer than 20% of all nodes are malicious. Blindspot [56] and `Drac` [46] suggest peering only with contacts from existing social networks, but this leaks information about users' relationships and results in small anonymity sets.

Onion routing. Works based on onion routing [51, 92, 93, 127], especially Tor [51], are widely adopted due to their relative low latency and ability to support millions of users. However, these systems are unable to resist traffic analysis attacks [68, 96, 112], even those performed by local adversaries [29, 78, 102, 132]. While future Internet architectures may address many of these shortcomings [34], we target a system that is deployable today.

DC networks. Another line of work is based on Dining Cryptographers (DC) networks [33, 42, 66]. They provide stronger guarantees than Pung under the same threat model, but they are peer-to-peer (requiring all users to know each other) and are based on all-to-all broadcast of messages. This results in high costs. Consequently, these systems typically accommodate only dozens of users. Verdict [43] and Dissent's successor [136] make great strides to reduce these costs and support thousands of users, but in the process introduce trusted infrastructure (under the anytrust model) which differs from our goals (§2).

Mailbox systems. Finally, there are a number of systems [25, 40, 41, 75, 79, 119, 130] that employ an architecture and techniques similar to Pung's (clients retrieve messages from per-round mailboxes kept at third-party servers). The key differences between these works and Pung is their reliance on at least one correct server, and the mechanisms that follow from that assumption. We elaborate on the most related ones below.

P^3 [75], like Pung, employs a key-value store from which users can privately pull messages. While P^3's focus is a retrieval mechanism that supports general queries when fetching a message (e.g., prefix search), Pung's primary goal is to drive down the cost of retrieval by introducing several batching optimizations (§4).

Riposte [41] targets a setting more fitting for whistle-blowers and informants where the sender wishes to remain anonymous from everyone (including all recipients). In contrast, Pung's goal is hiding the communication pattern between users who already know each other's identities. The Pynchon Gate [119] provides anonymity by composing a mix network with an IT-PIR scheme (§3.3). However, these guarantees hold only for passive adversaries who do not compromise mixes; under our threat model several attacks exist [100, 106, 107, 134]. Riffle [79] addresses this limitation by enhancing mixes with a verifiable shuffle, but retains the IT-PIR substrate and the anytrust model, which requires at least one correct server.

Vuvuzela [130] provides privacy through request shuffling and the careful addition of cover traffic rather than through PIR. Vuvuzela achieves significantly better performance than Pung (§7.2, §7.3), and it proposes an efficient dialing protocol, which Alpenhorn [80] enhances further. In contrast, Pung is not compatible with either dialing scheme, and we have not yet identified a suitable substitute (§5). However, Pung does introduce some benefits. In Vuvuzela, messages are ephemeral and can only be accessed during a single round; Pung supports long-lived messages that can be retrieved anytime prior to garbage collection (§6). Vuvuzela does not support group communications since it is based on point-to-point exchanges. Finally, the guarantees of a Vuvuzela deployment are based on differential privacy and are valid only for a certain number of rounds (based on a privacy budget). Pung's guarantees hold for any number of rounds.

10 Summary and conclusion

Our goal was to eliminate trust assumptions in private communication. To accomplish this goal, we leverage powerful cryptography and build Pung. Pung supports $10^3\times$ more users than prior systems in a similar threat model but falls short of systems that make trust assumptions. To improve performance, Pung targets a setting where clients retrieve multiple messages at once (§8). In this regime, Pung introduces new techniques that heavily amortize the costs of its cryptographic machinery. Our evaluation confirms that Pung reduces computational costs by up to $11\times$, at the expense of higher network costs. With these improvements, Pung presents an attractive design point for private communication systems.

Acknowledgments

Careful comments from Trinabh Gupta, Michael Lee, Josh Leners, Jay Lorch, Manos Kapritsos, Bryan Parno, Riad Wahby, the anonymous reviewers, and our shepherd Gernot Heiser made this paper better. We thank Michael Walfish for his thorough comments, which greatly improved this work. We also thank Carlos Aguilar Melchor for help with XPIR, and Frank McSherry for help with dataflow operators that Pung uses. Sebastian Angel was supported by NSF grants CNS-1055057 and CNS-1514422.

References

[1] Bleep. `http://www.bleep.pm`.

[2] ChatSecure. `https://chatsecure.org`.

[3] Open Whisper Systems. `https://whispersystems.org`.

[4] Slack: Be less busy. `https://slack.com/`.

[5] Telegram. `https://telegram.org`.

[6] Dissent: Provably anonymous overlay. `https://github.com/dedis/Dissent/tree/95f73`, Apr. 2010.

[7] Google says anything flowing across open WiFi is fair game. `https://goo.gl/fjOW2A`, Jan. 2014. Privacy SOS.

[8] Rust-crypto. `https://github.com/dagenix/rust-crypto/`, 2016.

[9] Vuvuzela: Private messaging system that hides metadata. `https://github.com/davidlazar/vuvuzela`, Sept. 2016.

[10] C. Aguilar-Melchor, J. Barrier, L. Fousse, and M.-O. Killijian. XPIR: Private information retrieval for everyone. In *Proceedings of the Privacy Enhancing Technologies Symposium (PETS)*, July 2016.

[11] C. Aguilar-Melchor, J. Barrier, L. Fousse, and M.-O. Killijian. XPIR: Private information retrieval for everyone. `https://github.com/xpir-team/xpir/`, 2016.

[12] S. Angel and S. Setty. Unobservable communication over fully untrusted infrastructure (extended version). Technical Report TR-16-16, The University of Texas at Austin, Oct. 2016.

[13] J. Angwin, C. Savage, J. Larson, H. Moltke, L. Poitras, and J. Risen. AT&T helped U.S. spy on Internet on a vast scale. `http://goo.gl/Jfsm18`, Aug. 2015. The New York Times.

[14] Y. Azar, A. Z. Broder, A. R. Karlin, and E. Upfal. Balanced allocations. In *Proceedings of the ACM Symposium on Theory of Computing (STOC)*, May 1994.

[15] J. Ball. GCHQ captured emails of journalists from top international media. `http://goo.gl/YzXnYK`, Jan. 2015. The Guardian.

[16] J. Bamford. Shady companies with ties to Israel wiretap the U.S. for the NSA. `http://goo.gl/bdi7w4`, Apr. 2012. Wired.

[17] A. Beimel and S. Dolev. Buses for anonymous message delivery. *Journal of Cryptology*, 16(1), Jan. 2003.

[18] A. Beimel, Y. Ishai, and T. Malkin. Reducing the servers computation in private information retrieval: PIR with preprocessing. In *Proceedings of the International Cryptology Conference (CRYPTO)*, Aug. 2000.

[19] M. Bellare, R. Canetti, and H. Krawczyk. Pseudorandom functions revisited: The cascade construction and its concrete security. In *Proceedings of the IEEE Symposium on Foundations of Computer Science (FOCS)*, Oct. 1996.

[20] M. Bellare and A. Lysyanskaya. Symmetric and dual PRFs from standard assumptions: A generic validation of an HMAC assumption. Cryptology ePrint Archive, Report 2015/1198, Dec. 2015. `http://eprint.iacr.org/2015/1198.pdf`.

[21] M. Bellare and C. Namprempre. Authenticated encryption: Relations among notions and analysis of the generic composition paradigm. In *International Conference on the Theory and Application of Cryptology and Information Security (ASIACRYPT)*, Dec. 2000.

[22] O. Berthold, H. Federrath, and S. Köpsell. Web MIXes: A system for anonymous and unobservable Internet access. In *Proceedings of the International Workshop on Designing Privacy Enhancing Technologies: Design Issues in Anonymity and Unobserbability*, July 2000.

[23] O. Berthold and H. Langos. Dummy traffic against long term intersection attacks. In *Proceedings of the Workshop on Privacy Enhancing Technologies (PET)*, Mar. 2002.

[24] B. H. Bloom. Space/time trade-offs in hash coding with allowable errors. *Communications of the ACM*, 13(7), July 1970.

[25] N. Borisov, G. Danezis, and I. Goldberg. DP5: A private presence service. In *Proceedings of the Privacy Enhancing Technologies Symposium (PETS)*, June 2015.

[26] J. Brickell and V. Shmatikov. Efficient anonymity-preserving data collection. In *Proceedings of the ACM SIGKDD Conference on Knowledge Discovery and Data Mining (KDD)*, Aug. 2006.

[27] S. Buttar. Dragnet NSA spying survives: 2015 in review. `https://goo.gl/JsNgS7`, Dec. 2015. Electronic Frontier Foundantion.

[28] C. Cachin, S. Micali, and M. Stadler. Computationally private information retrieval with polylogarithmic communication. In *Proceedings of the International Conference on the Theory and Applications of Cryptographic Techniques (EUROCRYPT)*, May 1999.

[29] X. Cai, X. C. Zhang, B. Joshi, and R. Johnson. Touching from a distance: Website fingerprinting attacks and defenses. In *Proceedings of the ACM Conference on Computer and Communications Security (CCS)*, Oct. 2008.

[30] Y.-C. Chang. Single database private information retrieval with logarithmic communication. In *Proceedings of the Australasian Conference on Information Security and Privacy*, July 2004.

[31] D. Chaum, F. Javani, A. Kate, A. Krasnova, J. de Ruiter, and A. T. Sherman. cMix: Anonymization by high-performance scalable mixing. Cryptology ePrint Archive, Report 2016/008, Jan. 2016. `http://eprint.iacr.org/2016/008.pdf`.

[32] D. L. Chaum. Untraceable electronic mail, return addresses, and digital pseudonyms. *Communications of the ACM*, 24(2), Feb. 1981.

[33] D. L. Chaum. The dining cryptographers problem: Unconditional sender and recipient untraceability. *Journal of Cryptology*, 1(1), 1988.

[34] C. Chen, D. E. Asoni, D. Barrera, G. Danezis, and A. Perrig. HORNET: High-speed onion routing at the network layer. In *Proceedings of the ACM Conference on Computer and Communications Security (CCS)*, Oct. 2015.

[35] B. Chor, N. Gilboa, and M. Naor. Private information retrieval by keywords. Cryptology ePrint Archive, Report 1998/003, Feb. 1998. http://eprint.iacr.org/1998/003.

[36] B. Chor, O. Goldreich, E. Kushilevitz, and M. Sudan. Private information retrieval. In *Proceedings of the IEEE Symposium on Foundations of Computer Science (FOCS)*, Oct. 1995.

[37] I. Clarke, O. Sandberg, B. Wiley, and T. W. Hong. Freenet: A distributed anonymous information storage and retrieval system. In *Proceedings of the International Workshop on Designing Privacy Enhancing Technologies: Design Issues in Anonymity and Unobserbability*, July 2000.

[38] D. Cole. We kill people based on metadata. http://goo.gl/LWKQLx, May 2014. The New York Review of Books.

[39] T. Cook. A message to our customers. http://www.apple.com/customer-letter/, Feb. 2016.

[40] D. A. Cooper and K. P. Birman. Preserving privacy in a network of mobile computers. In *Proceedings of the IEEE Symposium on Security and Privacy*, May 1995.

[41] H. Corrigan-Gibbs, D. Boneh, and D. Mazières. Riposte: An anonymous messaging system handling millions of users. In *Proceedings of the IEEE Symposium on Security and Privacy*, May 2015.

[42] H. Corrigan-Gibbs and B. Ford. Dissent: Accountable anonymous group messaging. In *Proceedings of the ACM Conference on Computer and Communications Security (CCS)*, Oct. 2010.

[43] H. Corrigan-Gibbs, D. I. Wolinsky, and B. Ford. Proactively accountable anonymous messaging in Verdict. In *Proceedings of the USENIX Security Symposium*, Aug. 2013.

[44] Council of Europe. European Convention on Human Rights: Article 8. http://www.echr.coe.int/Documents/Convention_ENG.pdf, Nov. 1950.

[45] G. Danezis, C. Diaz, and P. Syverson. Systems for anonymous communication. https://securewww.esat.kuleuven.be/cosic/publications/article-1335.pdf, Aug. 2009.

[46] G. Danezis, C. Diaz, C. Troncoso, and B. Laurie. Drac: An architecture for anonymous low-volume communications. In *Proceedings of the Privacy Enhancing Technologies Symposium (PETS)*, July 2010.

[47] G. Danezis, R. Dingledine, and N. Mathewson. Mixminion: Design of a type III anonymous remailer protocol. In *Proceedings of the IEEE Symposium on Security and Privacy*, May 2003.

[48] C. Devet, I. Goldberg, and N. Heninger. Optimally robust private information retrieval. In *Proceedings of the USENIX Security Symposium*, Aug. 2012.

[49] W. Diffie and M. E. Hellman. New directions in cryptography. *IEEE Transactions on Information Theory*, 22(6), Nov. 1976.

[50] R. Dingledine. Did the FBI pay a university to attack Tor users? https://goo.gl/NB3hSR, Nov. 2015. Tor Project.

[51] R. Dingledine, N. Mathewson, and P. Syverson. Tor: The second-generation onion router. In *Proceedings of the USENIX Security Symposium*, Aug. 2004.

[52] J. R. Douceur. The sybil attack. In *Proceedings of the International Workshop on Peer-to-Peer Systems*, Mar. 2002.

[53] M. Dworkin. Recommendation for block cipher modes of operation: Galois/Counter Mode (GCM) and GMAC. Technical Report SP 800-38D, National Institute of Standards and Technology, Nov. 2007.

[54] C. Egger, J. Schlumberger, C. Kruegel, and G. Vigna. Practical attacks against the I2P network. In *Proceedings of the International Symposium on Research in Attacks, Intrusions and Defenses (RAID)*, Nov. 2013.

[55] M. J. Freedman and R. Morris. Tarzan: A peer-to-peer anonymizing network layer. In *Proceedings of the ACM Conference on Computer and Communications Security (CCS)*, Nov. 2002.

[56] J. Gardiner and S. Nagaraja. Blindspot: Indistinguishable anonymous communications. arXiv:1408/0784v2, Aug. 2014. http://arxiv.org/abs/1408.0784.

[57] W. Gasarch and A. Yerukhimovich. Computationally inexpensive cPIR. https://www.cs.umd.edu/~arkady/papers/pirlattice.pdf, 2006.

[58] C. Gentry and Z. Ramzan. Single-database private information retrieval with constant communication rate. In *Proceedings of the International Colloquium on Automata, Languages and Programming (ICALP)*, July 2005.

[59] I. Goldberg. Improving the robustness of private information retrieval. In *Proceedings of the IEEE Symposium on Security and Privacy*, May 2007.

[60] P. Golle and A. Juels. Dining cryptographers revisited. In *Proceedings of the International Conference on the Theory and Applications of Cryptographic Techniques (EUROCRYPT)*, May 2004.

[61] A. Greenberg. Whatsapp just switched on end-to-end encryption for hundreds of millions of users. http://www.wired.com/2014/11/whatsapp-encrypted-messaging/, Nov. 2014.

[62] G. Greenwald and R. Gallagher. New Zealand launched mass surveillance project while publicly denying it. https://goo.gl/UwNpwV, Sept. 2014. The Intercept.

[63] G. Greenwald and E. MacAskill. NSA Prism program taps in to user data of Apple, Google and others. http://goo.gl/qETWUq, June 2013. The Guardian.

[64] J. Groth, A. Kiayias, and H. Lipmaa. Multi-query computationally-private information retrieval with constant communication rate. In *Proceedings of the International Conference on Practice and Theory in Public Key Cryptography (PKC)*, May 2010.

[65] C. Gülcü and G. Tsudik. Mixing E-mail with Babel. In *Proceedings of the Network and Distributed System Security Symposium (NDSS)*, Feb. 1996.

[66] E. Gün Sirer, S. Goel, M. Robson, and D. Engin. Eluding carnivores: File sharing with strong anonymity. In *Proceedings of the ACM SIGOPS European Workshop*, Sept. 2004.

[67] T. Gupta, N. Crooks, W. Mulhern, S. Setty, L. Alvisi, and M. Walfish. Scalable and private media consumption with Popcorn. In *Proceedings of the USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, Mar. 2016.

[68] N. Hopper, E. Y. Vasserman, and E. Chan-Tin. How much anonymity does network latency leak? In *Proceedings of the ACM Conference on Computer and Communications Security (CCS)*, Oct. 2007.

[69] Y. Ishai, E. Kushilevitz, R. Ostrovsky, and A. Sahai. Batch codes and their applications. In *Proceedings of the ACM Symposium on Theory of Computing (STOC)*, June 2004.

[70] J. Katz and M. Yung. Unforgeable encryption and chosen ciphertext secure modes of operation. In *Proceedings of the Fast Software Encryption Workshop (FSE)*, Apr. 2000.

[71] D. Kesdogan, D. Agrawal, V. Pham, and D. Rautenbach. Fundamental limits on the anonymity provided by the MIX technique. In *Proceedings of the IEEE Symposium on Security and Privacy*, May 2006.

[72] D. Kesdogan, J. Egner, and R. Büschkes. Stop-And-Go-MIXes providing probabilistic anonymity in an open system. In *Proceedings of the International Workshop on Information Hiding*, Apr. 1998.

[73] A. Kiayias, N. Leonardos, H. Lipmaa, K. Pavlyk, and Q. Tang. Optimal rate private information retrieval from homomorphic encryption. In *Proceedings of the Privacy Enhancing Technologies Symposium (PETS)*, July 2015.

[74] A. Kirsch and M. Mitzenmacher. Less hashing, same performance: Building a better Bloom filter. *Journal of Random Structures and Algorithms*, 33(2), Sept. 2008.

[75] L. Kissner, A. Oprea, M. K. Reiter, D. Song, and K. Yang. Private keyword-based push and pull with applications to anonymous communication. In *Proceedings of the International Conference on Applied Cryptography and Network Security (ACNS)*, June 2004.

[76] H. Krawczyk. Cryptographic extraction and key derivation: The HKDF scheme. In *Proceedings of the International Cryptology Conference (CRYPTO)*, Aug. 2010.

[77] E. Kushilevitz and R. Ostrovsky. Replication is not needed: Single database, computationally-private information retrieval. In *Proceedings of the IEEE Symposium on Foundations of Computer Science (FOCS)*, Oct. 1997.

[78] A. Kwon, M. AlSabah, D. Lazar, M. Dacier, and S. Devadas. Circuit fingerprinting attacks: Passive deanonymization of Tor hidden services. In *Proceedings of the USENIX Security Symposium*, Aug. 2015.

[79] A. Kwon, D. Lazar, S. Devadas, and B. Ford. Riffle: An efficient communication system with strong anonymity. In *Proceedings of the Privacy Enhancing Technologies Symposium (PETS)*, July 2016.

[80] D. Lazar and N. Zeldovich. Alpenhorn: Bootstrapping secure communication without leaking metadata. In *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, Nov. 2016.

[81] S. Le Blond, D. Choffnes, W. Caldwell, P. Druschel, and N. Merritt. Herd: A scalable, traffic analysis resistant anonymity network for VoIP systems. In *Proceedings of the ACM SIGCOMM Conference*, Aug. 2015.

[82] S. Le Blond, D. Choffnes, W. Zhou, P. Druschel, H. Ballani, and P. Francis. Towards efficient traffic-analysis resistant anonymity networks. In *Proceedings of the ACM SIGCOMM Conference*, Aug. 2013.

[83] R. Lenzner. ATT, Verizon, Sprint are paid cash by NSA for your private communications. `http://goo.gl/x7Cz1m`, Sept. 2013. Forbes.

[84] B. N. Levine, M. K. Reiter, C. Wang, and M. Wright. Timing attacks in low-latency mix systems. In *Proceedings of the International Financial Cryptography Conference*, Feb. 2004.

[85] M. Lewis. *Flash Boys: A Wall Street Revolt*. W.W. Norton & Company, Mar. 2014.

[86] W. Lueks and I. Goldberg. Sublinear scaling for multi-client private information retrieval. In *Proceedings of the International Financial Cryptography and Data Security Conference*, Jan. 2015.

[87] N. Mathewson and R. Dingledine. Practical traffic analysis: Extending and resisting statistical disclosure. In *Proceedings of the Workshop on Privacy Enhancing Technologies (PET)*, May 2004.

[88] J. Mayer, P. Mutchler, and J. C. Mitchell. Evaluating the privacy properties of telephone metadata. *Proceedings of the National Academy of Sciences of the United States of America (PNAS)*, 113(20), May 2016.

[89] F. McSherry. Timely dataflow. `https://github.com/frankmcsherry/timely-dataflow/`, 2016.

[90] J. Menn. Yahoo secretly scanned customer emails for U.S. intelligence. `https://goo.gl/KZuUYo`, Oct. 2016. Reuters.

[91] P. Mittal and N. Borisov. Information leaks in structured peer-to-peer anonymous communication systems. In *Proceedings of the ACM Conference on Computer and Communications Security (CCS)*, Oct. 2008.

[92] P. Mittal, F. Olumofin, C. Troncoso, N. Borisov, and I. Goldberg. PIR-Tor: Scalable anonymous communication using private information retrieval. In *Proceedings of the USENIX Security Symposium*, Aug. 2011.

[93] P. Mittal, M. Wright, and N. Borisov. Pisces: Anonymous communication using social networks. In *Proceedings of the Network and Distributed System Security Symposium (NDSS)*, Feb. 2013.

[94] M. Mitzenmacher. The power of two choices in randomized load balancing. *IEEE Transactions on Parallel and Distributed Systems*, 12(10), Oct. 2001.

[95] M. Mitzenmacher and E. Upfal. *Probability and Computing: Randomized Algorithms and Probabilistic Analysis*. Cambridge University Press, Jan. 2005.

[96] S. J. Murdoch and G. Danezis. Low-cost traffic analysis of Tor. In *Proceedings of the IEEE Symposium on Security and Privacy*, May 2005.

[97] D. G. Murray, F. McSherry, R. Isaacs, M. Isard, P. Barham, and M. Abadi. Naiad: A timely dataflow

system. In *Proceedings of the ACM Symposium on Operating Systems Principles (SOSP)*, Nov. 2013.

[98] A. Nambiar and M. Wright. Salsa: A structured approach to large-scale anonymity. In *Proceedings of the ACM Conference on Computer and Communications Security (CCS)*, Nov. 2006.

[99] M. Naor and M. Yung. Public-key cryptosystems provably secure against chosen ciphertext attacks. In *Proceedings of the ACM Symposium on Theory of Computing (STOC)*, May 1990.

[100] L. Nguyen and R. Safavi-Naini. Breaking and mending resilient mix-nets. In *Proceedings of the Workshop on Privacy Enhancing Technologies (PET)*, Mar. 2003.

[101] P. Paillier. Public-key cryptosystems based on composite degree residuosity classes. In *Proceedings of the International Conference on the Theory and Applications of Cryptographic Techniques (EUROCRYPT)*, May 1999.

[102] A. Panchenko, L. Niessen, A. Zinnen, and T. Engel. Website fingerprinting in onion routing based anonymization networks. In *Proceedings of the ACM Workshop on Privacy in the Electronic Society (WPES)*, Oct. 2011.

[103] D. C. Parkes, C. Thorpe, and W. Li. Achieving trust without disclosure: Dark pools and a role for secrecy-preserving verification. In *Proceedings of the Conference on Auctions, Market Mechanisms and Their Applications (AMMA)*, Aug. 2015.

[104] M. B. Paterson, D. R. Stinson, and R. Wei. Combinatorial batch codes. *Advances in Mathematics of Communications (AMC)*, 3(1), Feb. 2009.

[105] A. Pfitzmann and M. Hansen. A terminology for talking about privacy by data minimization: Anonymity, unlinkability, undetectability, unobservability, pseudonymity, and identity management. `http://dud.inf.tu-dresden.de/literatur/Anon_Terminology_v0.34.pdf`, Aug. 2010.

[106] B. Pfitzmann. Breaking an efficient anonymous channel. In *Proceedings of the International Conference on the Theory and Applications of Cryptographic Techniques (EUROCRYPT)*, May 1995.

[107] B. Pfitzmann and A. Pfitzmann. How to break the direct RSA-implementation of mixes. In *Proceedings of the International Conference on the Theory and Applications of Cryptographic Techniques (EUROCRYPT)*, Apr. 1989.

[108] A. Piotrowska, J. Hayes, N. Gelernter, G. Danezis, and A. Herzberg. AnoNotify: A private notification service. Cryptology ePrint Archive, Report 2016/466, May 2016. `http://eprint.iacr.org/2016/466.pdf`.

[109] E. Protalinski. Facebook scans chats and posts for criminal activity. `http://goo.gl/pfV9XE`, July 2012. CNET.

[110] C. Rackoff and D. R. Simon. Non-interactive zero-knowledge proof of knowledge and chosen ciphertext attack. In *Proceedings of the International Cryptology Conference (CRYPTO)*, Aug. 1991.

[111] A. S. Rawat, Z. Song, A. G. Dimakis, and A. Gál. Batch codes through dense graphs without short cycles. *IEEE Transactions on Information Theory*, 62(4), Apr. 2016.

[112] J.-F. Raymond. Traffic analysis: Protocols, attacks, design issues and open problems. In *Proceedings of the Workshop on Privacy Enhancing Technologies (PET)*, May 2001.

[113] M. K. Reiter and A. D. Rubin. Crowds: Anonymity for web transactions. *ACM Transactions on Information and System Security*, 1(1), Nov. 1998.

[114] M. Rennhard and B. Plattner. Introducing MorphMix: Peer-to-peer based anonymous Internet usage with collusion detection. In *Proceedings of the ACM Workshop on Privacy in the Electronic Society (WPES)*, Nov. 2002.

[115] R. L. Rivest. Chaffing and winnowing: Confidentiality without encryption. *CryptoBytes Technical Newsletter (RSA Laboratories)*, 4(1), July 1998.

[116] P. Rogaway. The moral character of cryptographic work. Cryptology ePrint Archive, Report 2015/1162, Dec. 2015. `http://eprint.iacr.org/2015/1162.pdf`.

[117] A. Rusbridger. The Snowden leaks and the public. `http://goo.gl/VOQL86`, Nov. 2013. The New York Review of Books.

[118] D. Rushe. Yahoo $250,000 daily fine over NSA data refusal was set to double 'every week'. `http://goo.gl/FZGfTT`, Sept. 2014. The Guardian.

[119] L. Sassaman, B. Cohen, and N. Mathewson. The Pynchon Gate: A secure method of pseudonymous mail retrieval. In *Proceedings of the ACM Workshop on Privacy in the Electronic Society (WPES)*, Nov. 2005.

[120] B. Schneier. *Data and Goliath: The Hidden Battles to Collect Your Data and Control Your World*. W.W. Norton & Company, Mar. 2015.

[121] R. Sherwood, B. Bhattacharjee, and A. Srinivasan. P5: A protocol for scalable anonymous communication. In *Proceedings of the IEEE Symposium on Security and Privacy*, May 2002.

[122] V. Shmatikov and M.-H. Wang. Timing analysis in low-latency mix networks: Attacks and defenses. In *Proceedings of the European Symposium on Research in Computer Security (ESORICS)*, Sept. 2006.

[123] N. Silberstein and A. Gál. Optimal combinatorial batch codes based on block designs. *Designs, Codes and Cryptography*, 78(2), Feb. 2016.

[124] A. Singh, T.-W. Ngan, P. Druschel, and D. S. Wallach. Eclipse attacks on overlay networks: Threats and defenses. In *Proceedings of the IEEE International Conference on Computer Communications (INFOCOM)*, Apr. 2006.

[125] R. Sion and B. Carbunar. On the computational practicality of private information retrieval. In *Proceedings of the Network and Distributed System Security Symposium (NDSS)*, Feb. 2007.

[126] V. Strassen. Gaussian elimination is not optimal. *Numerische Mathematik*, 13(4), Aug. 1969.

[127] P. F. Syverson, D. M. Goldschlag, and M. G. Reed. Anonymous connections and onion routing. In *Proceedings of the IEEE Symposium on Security and Privacy*, May 1997.

[128] United Nations General Assembly. The Universal

Declaration of Human Rights: Article 12. `http://www.un.org/en/universal-declaration-human-rights/`, Dec. 1948.

[129] United States Congress. Electronic Communications Privacy Act of 1986 (ECPA). `https://it.ojp.gov/privacyliberty/authorities/statutes/1285`, Oct. 1986.

[130] J. van den Hooff, D. Lazar, M. Zaharia, and N. Zeldovich. Vuvuzela: Scalable private messaging resistant to traffic analysis. In *Proceedings of the ACM Symposium on Operating Systems Principles (SOSP)*, Oct. 2015.

[131] M. Waidner and B. Pfitzmann. The dining cryptographers in the disco: Unconditional sender and recipient untraceability with computationally secure serviceability. In *Proceedings of the International Conference on the Theory and Applications of Cryptographic Techniques (EUROCRYPT)*, Apr. 1989.

[132] T. Wang and I. Goldberg. Improved website fingerprinting on Tor. In *Proceedings of the ACM Workshop on Privacy in the Electronic Society (WPES)*, Nov. 2013.

[133] S. Warren and L. Brandeis. The right to privacy. *Harvard Law Review*, 4(5), Dec. 1890.

[134] D. Wikström. Five practical attacks for “optimistic mixing for exit-polls”. In *Proceedings of the Conference on Selected Areas in Cryptography (SAC)*, Aug. 2003.

[135] P. Williams and R. Sion. Usable PIR. In *Proceedings of the Network and Distributed System Security Symposium (NDSS)*, Feb. 2008.

[136] D. I. Wolinsky, H. Corrigan-Gibbs, B. Ford, and A. Johnson. Dissent in numbers: Making strong anonymity scale. In *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, Oct. 2012.

[137] D. I. Wolinsky, H. Corrigan-Gibbs, B. Ford, and A. Johnson. Scalable anonymous group communication in the anytrust model. In *Proceedings of the European Workshop on System Security (EUROSEC)*, Apr. 2012.

[138] B. Zantout and R. A. Haraty. I2P data communication system. In *Proceedings of the International Conference on Networks*, Jan. 2011.

Alpenhorn: Bootstrapping Secure Communication without Leaking Metadata

David Lazar and Nickolai Zeldovich
MIT CSAIL

Abstract

Alpenhorn is the first system for initiating an encrypted connection between two users that provides strong *privacy* and *forward secrecy* guarantees for metadata (i.e., information about which users connected to each other) and that does not require out-of-band communication other than knowing the other user's Alpenhorn username (email address). This resolves a significant shortcoming in all prior works on private messaging, which assume an out-of-band key distribution mechanism.

Alpenhorn's design builds on three ideas. First, Alpenhorn provides each user with an address book of friends that the user can call. Second, when a user adds a friend for the first time, Alpenhorn ensures the adversary does not learn the friend's identity, by using *identity-based encryption* in a novel way to privately determine the friend's public key. Finally, when calling a friend, Alpenhorn ensures forward secrecy of metadata by storing pairwise shared secrets in friends' address books, and evolving them over time, using a new *keywheel* construction. Alpenhorn relies on a number of servers, but operates in an anytrust model, requiring just one of the servers to be honest.

We implemented a prototype of Alpenhorn, and integrated it into the Vuvuzela private messaging system (which did not previously provide privacy or forward secrecy of metadata when initiating conversations). Experimental results show that Alpenhorn can scale to many users, supporting 10 million users on three Alpenhorn servers with an average dial latency of 150 seconds and a client bandwidth overhead of 3.7 KB/sec.

1 Introduction

To achieve privacy in a communication system, it is not enough to just hide the contents of the messages sent or received by a user. It is also important to hide who the user is communicating with, at what time they are communicating, and whether they are communicating with anyone at all; we refer to such information as *metadata*. For instance, researchers have shown that they can learn significant amounts of sensitive information by looking at just what phone numbers a person called [33], who the person emailed, their IP address, or social network connections [18]. Similarly, NSA officials have also said that metadata is crucial for surveillance [38].

Recent work shows that it is possible to build private messaging systems that hide metadata at scale [10, 15, 19, 28, 29, 41, 43]. Unfortunately, these systems do not provide users with a convenient way to bootstrap communication without leaking metadata in the process. This impedes practical deployment and precludes any end-to-end metadata privacy guarantees.

Alpenhorn is the first system to address this problem. Functionally, Alpenhorn allows users to initiate a conversation: that is, Alice can use Alpenhorn to call Bob, and Alpenhorn will ensure that Bob knows that Alice is calling, and that Alice and Bob agree on a fresh cryptographic key, called a session key, to protect their conversation. Alpenhorn is purely a bootstrapping protocol: the actual conversation can take place through one of the systems mentioned earlier. Crucially, Alpenhorn provides *privacy* and *forward secrecy* of metadata. This means that an adversary cannot determine who, if anyone, a user might be calling at any given time, and even if the adversary later compromises a user's computer, they will not be able to tell what calls the user made or received in the past.

To understand the challenges faced by Alpenhorn, consider the traditional approach for establishing a session key between users, which works in two steps. First, users learn of each other's long-term public keys, through some public key infrastructure (PKI) system. In the second step, users run a key exchange protocol, such as Diffie-Hellman, to establish a fresh session key, and they use their long-term keys to confirm each other's identity. These two steps correspond to the two challenges faced by Alpenhorn:

First, looking up a user's public key can leak metadata in itself. For instance, if Alice asks a key server for Bob's public key, and the adversary learns about this request, the adversary now knows Alice is about to call Bob. This violates Alpenhorn's goal of achieving *privacy* for metadata, and most existing PKI systems operate in this manner.

Second, even if users somehow manage to obtain each others' public keys, long-term public keys are a poor fit for metadata *forward secrecy*. Specifically, key exchange protocols like Diffie-Hellman authenticate participants by signing messages, which makes it obvious to an adversary who the participants are. A strawman solution is to encrypt these messages using the other user's public key, and to broadcast these messages, so that an adversary cannot tell who the intended recipient is. Even ignoring the performance overheads, this strawman fails to provide forward secrecy, because any adversary that later compromises the recipient's computer will obtain that user's long-term private key, and will be able to learn about all past incoming calls received by that user, by decrypting those messages.

Alpenhorn addresses these challenges using three ideas. First, instead of using long-term public keys to encrypt key exchange messages, Alpenhorn maintains an address book on each user's computer, containing a pairwise shared secret for each of that user's friends. This helps ensure forward secrecy because there is no long-term encryption key for an adversary to compromise.

Second, to allow users to add friends to their address book, Alpenhorn uses identity-based encryption (IBE) [7, 17, 39]. IBE is different from traditional public-key cryptography, in that a user's *public* key is purely a mathematical function of their username, such as an email address, together with a master public key from some server.[1] This allows Alpenhorn to compute a friend's public key without leaking the friend's identity. As described in §4, Alpenhorn extends IBE to handle server compromises and to ensure forward secrecy.

Finally, Alpenhorn must also provide privacy and forward secrecy for the metadata involved in actually initiating a conversation. To do this, Alpenhorn uses a novel *keywheel* construction, which continuously evolves all shared secrets in a user's address book, so as to provide forward secrecy while still ensuring that, at any given time, two friends have the same secret value in their address books.

Alpenhorn relies on two sets of servers: the IBE servers, mentioned above, and a set of mixnet servers, whose job is to hide the source of every message. Both the IBE and mixnet servers operate in the *anytrust* model, requiring just one honest server for security. The use of a trusted server allows Alpenhorn to achieve good performance, compared to purely cryptographic approaches like private information retrieval that do not trust any servers at all. §3 describes Alpenhorn's precise guarantees and assumptions.

To evaluate Alpenhorn, we implemented a prototype in Go, and integrated it with the Vuvuzela private messaging system, which did not previously provide privacy or forward secrecy for bootstrapping conversations. Integrating Alpenhorn into applications is straightforward; modifying Vuvuzela to use Alpenhorn required changing 200 lines of code. Alpenhorn's performance scales well with the number of users: 3 Alpenhorn servers can support 10 million users with 5% of them initiating a conversation every 5 minutes, with a modest client-side bandwidth cost of 3.7 KB/sec. The client-side overhead in particular is $\sim 10\times$ less than that of the equivalent dialing protocol in Vuvuzela (which fails to provide the same privacy guarantees).

[1]To achieve this, the user's *private* key must be generated by a server holding the corresponding master secret key.

In summary, the contributions of this paper are:

- Alpenhorn, the first system for establishing session keys that provides privacy and forward secrecy for metadata;
- a novel way of using IBE in an anytrust setting to achieve metadata forward secrecy;
- the keywheel construction, which allows the Alpenhorn client to establish fresh session keys with low latency and low bandwidth overheads;
- a prototype implementation of Alpenhorn; and
- an experimental evaluation of Alpenhorn that demonstrates it can scale to 10 million users.

2 Related work

A common way to bootstrap private messaging is to assume that users have exchanged keys or secrets out-of-band. For example, the Ricochet [13] private messaging system requires a user to know the other person's Ricochet ID (a public key) to start a conversation. The Pond [29] private messaging system uses a protocol called PANDA [2] to establish relationships between users that have previously shared a secret. In contrast, Alpenhorn allows two users to start a conversation without knowing each other's public keys or having a shared secret. Alpenhorn can be used to bootstrap PANDA (see §8.5).

Both Ricochet and Pond use Tor's hidden services [20]. Alpenhorn's privacy guarantees are stronger than those of Tor's hidden services in two ways. First, hidden services do not protect against traffic analysis. This is because Tor has many ways for an adversary to infer information based on traffic patterns. Alpenhorn uses techniques from Vuvuzela [41] to defeat traffic analysis (achieving differential privacy). Second, hidden services have a weaker adversary model for protecting metadata: e.g., an adversary that compromises the rendezvous point of a hidden service learns when that user is receiving calls. In contrast, Alpenhorn provides metadata privacy under an anytrust assumption (any $N-1$ out of N servers can be compromised).

DP5 [10] solves a related problem of *online presence*. It enables users to query their friends' online status (and learn additional information, such as their current IP address) without revealing metadata. DP5 assumes that every user already has a list of all of his friends and their public keys. This is precisely the problem that Alpenhorn is designed to address: to allow users to add new friends without knowing their public key, and to inform a user that someone wants to add them as a friend (or wants to call them).

Identity-based encryption (IBE). Alpenhorn uses IBE to exchange keys between two users for the first time. IBE typically assumes a trusted server known as the private key generator (PKG) that distributes private keys to users. To avoid trusting a single server, Boneh and Franklin [7] proposed using a distributed key generation (DKG) scheme to distribute the master secret key among multiple PKGs. Recently proposed DKG schemes require $3t + 1$ or $2t + 1$ servers to tolerate t dishonest servers, depending on the communication model [25]. Our Anytrust-IBE approach to distributing the PKG requires only 1 honest server, but this comes at the expense of availability (Alpenhorn provides no fault tolerance). In future work, we hope to explore whether more sophisticated cryptographic constructions can further improve Alpenhorn's performance [16].

Private information retrieval (PIR) could, in principle, provide an alternative to IBE for privately obtaining a user's public key [24]. To ensure forward secrecy, each user would need to periodically generate fresh keys, and upload them to a central database. Each user would also need to perform fake PIR queries even if they are not interested in looking up a key, to avoid leaking at what times the user is starting conversations. In practice, state-of-the-art PIR implementations cannot handle tens of millions of users each performing a query on a database containing tens of millions of records. Most implementations require quadratic [1, 28], or nearly quadratic [31] cost to handle N queries on a database containing N items. In contrast, Alpenhorn's design achieves a total server cost that is linear in the number of users, which enables it to support tens of millions of users. Alpenhorn's overall design also addresses an important challenge not faced by PIR: informing a user that someone wants to call them, and minimizing the bandwidth required for this notification.

Forward secrecy. IBE can achieve forward secrecy by having users generate a different key for each day (e.g., by concatenating the date with their username [7], or by using more efficient constructions [6]). A user can erase old private keys so that they are not disclosed when an adversary compromises the user's computer. However, this assumes that the IBE PKG server is not compromised: a compromised PKG could re-compute all old user private keys. In contrast, in Alpenhorn's design, even if an adversary compromises *all* PKGs, the adversary cannot decrypt past messages.

Binary tree encryption (BTE) [14] also allows users to forget old private keys to achieve forward secrecy. BTE does not require any interaction or trusted servers, but it also does not address two of the key problems faced by Alpenhorn: obtaining public keys without leaking metadata, and informing a user that someone has added them as a friend. Another downside of BTE is that the keys are much larger than in traditional public-key encryption.

The double ratchet algorithm [36] used by Signal and WhatsApp [42] continuously rotates session keys between users for forward secrecy, similar to Alpenhorn's keywheel. The key difference is that the ratchet ensures forward secrecy for *data*, whereas the keywheel produces dialing tokens, which Alpenhorn uses to ensure forward secrecy of *metadata*. Off-the-record messaging (OTR) [9] similarly rotates keys to achieve forward secrecy for data but does not hide metadata. Alpenhorn uses Bloom filters [5] to encode the dialing tokens produced by the keywheel, similar to the approach taken by AnoNotify [37].

Functions provided by the Alpenhorn library

```
// Initialize an Alpenhorn account
func Register(email string)

// Get your long-term key to share with friends
func MySigningKey() PublicKey

// Send friend request to the given email address.
// Their public key for extra verification is optional.
func AddFriend(email string, theirSigningKey *PublicKey)

// Call a friend; returns a shared secret known only
// to you and the friend. The call's intent is optional.
func Call(email string, intent int) SessionKey
```

Callbacks that must be implemented by the application

```
// This function is called when the client gets a friend
// request. Return true to accept the friend request.
func NewFriend(email string, theirSigningKey PublicKey) bool

// This function is invoked when client receives a call.
func IncomingCall(email string, intent int, key SessionKey)
```

Figure 1: Simplified Alpenhorn API.

3 Overview

To use Alpenhorn, the developer of a messaging application must integrate their application with the Alpenhorn client library, and specify a set of Alpenhorn servers that the library should use.[2] The API provided by the client library is shown in Figure 1. The API allows applications to perform two main tasks: to add a friend (for when the user asks the application to add a friend to their address book), and to initiate a call with a friend (for when the user asks the application to call a friend). Alpenhorn uses email addresses to identify users; §4 discusses what happens if an email server is compromised.

When a user starts the messaging application for the first time, the application calls `Register()`, passing in the user's own email address. `Register()` will generate a long-term signing key for the user, and register it with Alpenhorn. The user will have to prove their identity to the PKGs through confirmation emails.

Users can add friends by invoking the `AddFriend()` function. For example, if Alice and Bob are both us-

[2]For simplicity, this paper assumes the application developer sets up the Alpenhorn servers. Multiple applications can also share a set of Alpenhorn servers, but this paper does not discuss the issues that are involved in doing so.

ing an Alpenhorn-based messaging application, and Alice wants to add Bob as a friend, the application will call `AddFriend("bob@gmail.com", nil)`. In this example, Alice did not have any prior knowledge of Bob's long-term signing key, so the second argument is `nil`. However, Alice must know Bob's email address ahead of time; otherwise, there is no way for Alice to tell Alpenhorn whom she wants to add as a friend.

On the other side of the world, Bob's application receives a callback from the Alpenhorn library, `NewFriend("alice@gmail.com", "e27scvh08m...")`, and displays the request to Bob. If Bob had out-of-band knowledge of Alice's signing public key, he could verify it before accepting the request; an application can obtain the user's own long-term signing key by calling `MyPublicKey()`. Bob doesn't know Alice's long-term signing key, but he knows that she recently registered her email address with Alpenhorn, so he accepts the request knowing that the Alpenhorn servers have validated her identity.

The application returns `true` from the `NewFriend` callback to indicate to the Alpenhorn library that Bob accepted the friend request. Internally, this causes the library to send a friend request back to Alice to confirm the request.

After some time, Alice gets back the friend request from Bob, which confirms that she is now friends with him. At this point, Alice and Bob's Alpenhorn libraries have internally agreed on a shared secret, stored in their keywheels, and the Alpenhorn library continuously rolls forward this shared secret; however, this secret value is not directly exposed to the application.

The next day, Alice opens a chat window for Bob in her messaging app, which causes the application to invoke `Call("bob@gmail.com", 0)`. The second argument, `0`, is an application-specific *intent* that is passed along to the application on the other side; we discuss intents more in §5.3. In Alice's client, the Alpenhorn library returns a fresh shared key that Alice's application should use for the conversation, such as `"3xdq9t7vP0..."`. Shortly afterward, Bob's Alpenhorn library invokes the `IncomingCall("alice@gmail.com", 0, "3xdq9t7vP0...")` callback, and the application tells Bob about an incoming call from Alice. If he accepts, Alice and Bob can start talking to each other through the application's private messaging protocol, using the fresh key `"3xdq9t7vP0..."`.

3.1 Overall design

Figure 2 shows the major components of Alpenhorn. Each Alpenhorn client maintains a long-term signing key, described above, and an address book, consisting primarily of a keywheel table, which stores and rolls forward shared secrets with each of that user's friends. In addition to the client library, Alpenhorn relies on two sets of servers:

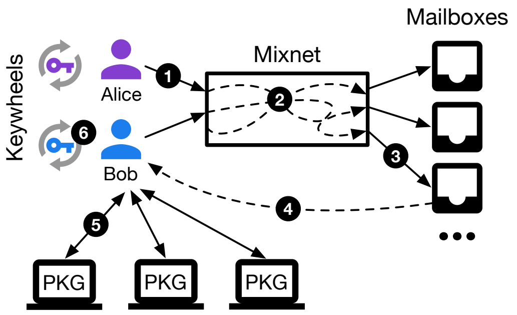


Figure 2: Overview of what happens when Bob adds Alice as a friend using Alpenhorn's add-friend protocol.

a set of *private-key generator* (PKG) servers, used for identity-based encryption, and a set of *mixnet* servers, used to hide which client submitted which request.

Alpenhorn consists of two protocols: the *add-friend* protocol for adding a friend to an address book, given their email address, and the *dialing* protocol for establishing a new conversation with a friend, which we describe in more detail in §4 and §5, respectively. This split allows Alpenhorn to achieve good performance. The add-friend protocol uses public-key cryptography, which is necessary to bootstrap communication with a new friend, but is relatively expensive (and thus has a higher latency). The dialing protocol uses symmetric-key cryptography, which allows existing friends to perform low-latency key exchanges. Figure 2 shows the add-friend protocol, which we will now describe; the dialing protocol is similar.

Alpenhorn clients send requests in periodic *rounds*, which are coordinated by the first mixnet server. Each client submits a fixed-size request to the mixnet in every round, shown by step 1, even if they don't want to add a friend at that moment. This provides *cover traffic*, so that an adversary cannot learn anything about who the user might be communicating with from the fact that a client is sending messages to Alpenhorn servers.

Requests are encrypted for the intended recipient, so that an adversary cannot decrypt the request's contents without the recipient's private key. The caller obtains the recipient's public key using identity-based encryption, which allows the client to obtain a given recipient's public key by simply computing it, without having to query any server for it. To ensure forward secrecy, the recipient's public key changes each round, and the recipient's client deletes each round's private key at the end of the round.

In step 2, the mixnet shuffles the requests for a given round, and adds additional *noise* to mask any statistical information that an adversary might learn at the end of the mixnet. The mixnet operates in an anytrust model; just one honest mixnet server is sufficient to provide security. Alpenhorn uses the Vuvuzela mixnet design [41], which adds enough noise to achieve differential privacy.

At the end of the mixnet, shown by step 3, client requests are distributed into *mailboxes* based on the intended recipient of the request. The request includes the destination mailbox ID in plaintext form for this purpose; it is computed by the client as the hash of the recipient's email address modulo the number of mailboxes; many users share the same mailbox. A special mailbox ID is used for cover traffic, so that it need not be processed further.

Each client then downloads their mailbox, in step 4. In step 5, the client contacts every PKG server to obtain its private key for this round. Alpenhorn combines the private keys from all PKG servers to ensure security as long as just one of them is honest. Then in step 6, the client tries to decrypt every request in the mailbox using the private keys for this round. If the decryption succeeds, the Alpenhorn client processes the incoming add-friend request, adds the resulting key to its keywheel, and sends an acknowledgment back (as another add-friend request). If the decryption fails, the request must have been intended for someone else, or was noise.

The dialing protocol works similarly, but significantly reduces the size of the mailbox using a Bloom filter [5] to efficiently encode a set of values submitted by clients. §5 describes the dialing protocol in more detail.

3.2 Security goals

Alpenhorn's security goals are motivated by the private messaging applications that Alpenhorn is aiming to support, such as Vuvuzela [41], Pung [1], and Pond [29]; specifically, Alpenhorn's guarantees should meet or exceed those of the application itself. Alpenhorn focuses on privacy, and does not achieve fault tolerance (a single server can make Alpenhorn unavailable). Specifically, Alpenhorn's guarantees are as follows:

Authenticated key exchange. A powerful adversary, capable of compromising servers and tampering with traffic, must not be able to learn the session keys generated by Alpenhorn. Alpenhorn must also prevent the adversary from impersonating other users, meaning the adversary should not be able to send friend requests or calls on behalf of an email address the adversary does not own.

Privacy for metadata. Alpenhorn should not reveal metadata about friends or calls (i.e., whom, if anyone, you call or add as a friend, or who, if anyone, calls you or adds you as a friend) even after the application has been running for a long time. Specifically, Alpenhorn provides differential privacy for this metadata, as formalized in Vuvuzela [41].

Forward secrecy for metadata. If the secret state of a server or client is compromised, the adversary must not be able learn metadata or the contents of messages sent in the past. An adversary can store all past traffic, in the hope of one day acquiring the private key of a server or client, so providing forward secrecy means that encryption keys must be short-lived and erased quickly after use.

An adversary that compromises a user's computer can, of course, obtain the contents of the address book from the user's chat application. This would allow the adversary to learn about a set of friends that the user may have talked to. If the user is concerned about this, they can remove a friend from their address book, at which point Alpenhorn's guarantees would prevent the adversary from determining if these two users were or were not friends in the past.

Worst-case security. If all servers are compromised, Alpenhorn is unable to offer privacy or forward secrecy for metadata. Nonetheless, Alpenhorn provides at least the same security guarantees as existing key-exchange protocols, even if all servers are compromised; specifically:

- If users have out-of-band knowledge of each other's public keys, Alpenhorn's API can use them to defeat man-in-the-middle attacks, as in existing protocols.
- Alpenhorn's client uses an SSH-like trust-on-first-use (TOFU) approach if out-of-band keys are not provided, by remembering the friend's long-term signing key from their first add-friend request. If two users called `AddFriend` when at least one server was honest, then a later compromise of all servers does not allow an adversary to mount a man-in-the-middle attack.
- In the absence of out-of-band keys, Alpenhorn could require each user to register their public key in a verifiable ledger (such as Keybase [26] or Namecoin [35]), and to send a proof to new friends that their key is registered in such a ledger. Depending on the scenario, this can prevent man-in-the-middle attacks or allow a user to detect that someone is impersonating them; we have not implemented this in our Alpenhorn prototype.
- If a client's state is compromised, then future interaction with that client is compromised. The user can recover by revoking all of his friendships and sending a new `AddFriend` request to each of his friends.

3.3 Threat model

Alpenhorn assumes an adversary that controls all but one of the Alpenhorn mixnet servers and all but one of the PKG servers (users need not know which ones), controls an arbitrary number of clients, and can monitor, block, delay, or inject traffic on any network link. Alpenhorn assumes that the client machines of legitimate users are not compromised, and that the client software properly implements the Alpenhorn protocol. Alpenhorn does not protect against malicious servers mounting denial-of-service attacks, but it is resilient to client denial-of-service attacks.

For forward secrecy, Alpenhorn assumes that the Alpenhorn client can irrevocably delete data from memory or disk (e.g., a cryptographic key or an address book entry). Forward secrecy guarantees could be subverted on short time scales by cold boot attacks [22], and on longer time scales by a storage system on the user's computer that allows recovering previously erased data (e.g., an SSD that does not overwrite data in place). Alpenhorn servers must also be able to securely erase memory, but they never store encryption keys on disk.

We make standard cryptographic assumptions like secure public and symmetric key encryption, Diffie-Hellman key-exchange, signature schemes, and hash functions. We also assume the security of pairing-based cryptography[3] which we use for identity-based encryption in §4.

We assume that the long-term signing public keys of the Alpenhorn servers are known to all users. These keys can be distributed in the Alpenhorn software package, similar to how web browsers ship with a list of CA keys.

Alpenhorn uses email addresses to identify users, and thus relies on the user's email provider for bootstrapping user identity. This boils down to two assumptions. First, when user A adds user B as a friend, A should know B's email address, and should be sure that B successfully registered for an Alpenhorn account under B's email address. Second, each user must periodically connect to Alpenhorn (at least once every 30 days) to prevent their Alpenhorn account from being reset by an adversary that may have compromised the user's email account. §4.6 describes Alpenhorn's use of email in more detail.

4 Add-friend protocol

When Alice adds Bob as a friend using the API shown in Figure 1, her client constructs a friend request that contains her email address, her public key, some signatures, and other sensitive data (as shown in Figure 3). Since the friend request contains sensitive information, Alice needs to encrypt it so that only Bob can read it. However, Alice does not have Bob's public key, and she can't ask a server to look it up, because that would leak metadata.

The add-friend protocol uses identity-based encryption (IBE) to enable Alice to encrypt her friend request using Bob's email address as the public key, as explained in §4.1. Since Alice already knows Bob's email address, this approach does not require any directory lookup and thus leaks no metadata. However, IBE traditionally assumes a trusted server to distribute private keys, which does not align with Alpenhorn's goals. We describe how we distribute the trust among multiple servers in §4.2.

Using IBE, Alice proceeds to encrypt the request, and sends it to a shared mailbox, which is a publicly known memory location on one of the Alpenhorn servers. The contents of this mailbox are visible to the servers and available for all clients to download, so Alpenhorn must ensure that Alice's client does not reveal any metadata in the process of placing the request in the mailbox.

[3] Chen et al [16] discusses the assumptions behind pairing-based cryptography; it has been deployed in systems such as Zerocash [4].

First, Alpenhorn must ensure that the encrypted friend requests in the mailbox do not reveal the email addresses of recipients for which they are encrypted. This property is known as *ciphertext anonymity* and is discussed in §4.3. Second, the keys used to produce the encrypted friend requests must be destroyed quickly. Otherwise, an attacker will keep a copy of the mailbox contents indefinitely, in hopes of one day compromising the private keys. This property is known as *forward secrecy* and is discussed in §4.4. Finally, we must prevent the adversary from learning who sent the friend requests, even in the face of sophisticated attacks like traffic analysis or tracking patterns in the number of messages in a mailbox. We borrow techniques from prior work on metadata privacy to ensure that an adversary does not learn who is friending whom, by adding noise messages to the mailbox [41], as discussed in §6.

Bob's client eventually downloads the mailbox from the server, containing encrypted friend requests. Bob also obtains his private key for that round from the private key generators (PKGs). Using his private key, Bob's client attempts to decrypt each friend request in the mailbox. When the client succeeds in decrypting one, the request is validated using the protocol in §4.5, and Bob is prompted to accept the request. If Bob accepts, his client sends a friend request back to Alice as an acknowledgment.

4.1 Identity-based encryption

Identity-based encryption (IBE) is a relatively new cryptographic primitive in which any username string (such as an email address) can be used as a public key. Typically, IBE assumes a single trusted party that knows everyone's private key, but we will see shortly that Alpenhorn distributes trust among many independent servers and that only one of these servers must remain honest.

For now, suppose there is a trusted server known as the private key generator (PKG). The PKG has a master public key M_{pub} known to all, and a master secret key M_{priv} known only to itself. An IBE scheme provides the following functions:

- Encrypt($M_{\text{pub}}, \textit{identity}, \textit{msg}$) $\rightarrow$ *ctxt*
  which encrypts a message to some identity string (e.g., a username, email address, or other unique identifier).
- Decrypt($\textit{identity}_{\text{priv}}, \textit{ctxt}$) $\rightarrow$ ($\textit{msg}, \textit{ok}$)
  which decrypts a ciphertext using the private key corresponding to some identity string.
- Extract($\textit{identity}, M_{\text{priv}}$) $\rightarrow$ $\textit{identity}_{\text{priv}}$
  which computes the private key corresponding to some identity string.

The PKG verifies the identities of users and issues to them private keys corresponding to their identities. For example, suppose the PKG identifies users by their email address. When Bob asks the PKG for the private key corresponding to "`bob@gmail.com`", the PKG can send a random nonce to that email address. If Bob can produce that nonce, the PKG gives him the private key for that email address. In practice, Alpenhorn uses a more secure scheme to authenticate email addresses, described in §4.6.

IBE's power comes from the fact that anyone with M_{pub} can encrypt a message to another user without any directory lookups, as long as they know the recipient's identity. Avoiding communication for looking up the recipient's public key avoids the possibility of that communication being intercepted by an adversary to learn metadata.

4.2 Distributing trust

Using a single trusted PKG means that if it were compromised, the adversary would be able to compute the IBE private keys of every user. To avoid this, Alpenhorn uses multiple independent PKGs. A naïve approach would be to onion-encrypt a message using the master public key of each PKG in turn. For example, suppose there are n PKGs with master public keys $M^1_{\text{pub}} \dots M^n_{\text{pub}}$. To encrypt an add-friend message for Bob, Alice could compute:

$$\begin{aligned}&\text{Encrypt}(M^1_{\text{pub}}, \text{“}\texttt{bob@gmail.com}\text{”},\\ &\quad \text{Encrypt}(M^2_{\text{pub}}, \text{“}\texttt{bob@gmail.com}\text{”}, \cdots\\ &\qquad\quad \text{Encrypt}(M^n_{\text{pub}}, \text{“}\texttt{bob@gmail.com}\text{”}, msg)\cdots))\end{aligned}$$

To decrypt this ciphertext, Bob must obtain the private key for his email address from each of the n PKGs. Now, even if many PKGs are compromised, the ciphertext stays private as long as one of the master secret keys (and Bob's corresponding private key) is unknown to the adversary. Although this would achieve Alpenhorn's security goals, it is inefficient because each layer of encryption adds additional space overhead, and because Bob's decryption takes time proportional to the number of PKGs.

Alpenhorn introduces a more efficient scheme, called Anytrust-IBE, that achieves the same goal of distributing the trust among n PKG servers, by adding together the master public keys in the Boneh-Franklin IBE scheme [7]:

$$\text{Encrypt}(\sum_{i=1}^{n} M^i_{\text{pub}}, \text{“}\texttt{bob@gmail.com}\text{”}, msg)$$

Bob can decrypt this ciphertext by adding his private keys:

$$\text{Decrypt}(\sum_{i=1}^{n} identity^i_{\text{priv}}, ctxt)$$

This scheme is efficient: once Bob obtains and adds up his private keys, neither the ciphertext size nor the decryption time depend on the number of PKGs. A technical report [30: §A] provides more details and a proof of security for Anytrust-IBE.

```
type FriendRequest struct {
    SenderEmail  string
    SenderKey    SigningKey
    SenderSig    Signature
    PKGSigs      MultiSignature
    DialingKey   DiffieHellmanKey
    DialingRound int
}
```

Figure 3: Alpenhorn friend request.

4.3 Ciphertext anonymity

To avoid leaking metadata, it is important that encrypted friend requests (produced by the IBE Encrypt function) do not reveal the intended recipient. This property is known as *ciphertext anonymity* [11], and it is not generally true of IBE schemes [12]. Alpenhorn deliberately uses the Boneh-Franklin IBE scheme [7] because it is one of the few IBE schemes with this property. Ciphertext anonymity is also necessary to generate noise messages in the mixnet (§6).

4.4 Forward secrecy

The encrypted friend requests created by the add-friend protocol eventually become public, so it is crucial that the keys to the ciphertext are destroyed quickly to limit the possibility of compromise. For IBE ciphertexts, there are two keys we worry about: the identity private keys held by users and the master secret keys held by the PKGs. When both sets of private keys are destroyed, ciphertexts created with the corresponding public keys become useless to the adversary. Better yet, in Anytrust-IBE, only one (rather than all) of the PKGs must destroy its master secret key to achieve forward secrecy.

The add-friend protocol operates in rounds to achieve forward secrecy. Every round, the PKGs create new master keys, and broadcast the public keys for that round to the users. Users must then obtain their private keys for that round from each PKG. After a preconfigured amount of time or after all users have obtained their private keys, each PKG deletes its master secret key for the round. Users also delete their identity private keys after downloading and scanning their mailbox for friend requests.

4.5 Authenticating requests

Figure 3 shows the structure of a friend request, obtained after decrypting a ciphertext in the add-friend mailbox. The request includes the sender's email address (e.g., "`alice@gmail.com`"), but how does the Bob the recipient verify that the friend request really came from Alice?

To authenticate the request, Alpenhorn's includes two signatures in the friend message. First, the `SenderSig` is a signature over the entire friend request using the sender's long-term signing key, `SenderKey`. If Bob happens to somehow know Alice's long-term key (e.g., because he got Alice's business card, which lists her signing public key), he can verify the authenticity of the message by verifying SenderSig using the key he obtained out-of-band.

If Bob does not know Alice's long-term signing key, Bob's client can rely on the PKGs to authenticate Alice. Specifically, when a user's client acquires the user's IBE private key, each PKG also responds with a signature of the user's long-term key and email address. The friend request includes a multi-signature [8], PKGSigs, which combines the signatures from all PKGs into a single compact value. Bob can check PKGSigs to ensure that SenderKey belongs to SenderEmail, as long as at least one PKG is honest.

4.6 Registering email addresses

Every round, users must authenticate to the PKG in order to obtain their private keys. To avoid manual user involvement at every round, Alpenhorn splits authentication into two steps: first, a manual account registration step, and second, an automatic private-key-generation step. The second (key-generation) step is straightforward: each PKG keeps track of the long-term signing key for every registered email address, and users can obtain their IBE private key for a round by signing a request with their long-term signing private key. The first registration step, however, is more complicated because it involves trusting the user's email account provider to bootstrap the user's identity.

When using Alpenhorn for the first time, Alice registers her email address with her long-term signing key at each PKG. Each PKG sends Alice a confirmation email containing a secret token, which Alice must send to the PKG to finish the registration.[4] After registration, each PKG locks the user's email address to that user's long-term signing key, to prevent anyone else (e.g., a malicious email provider) from re-registering the address.

There is no *quick* way to reset an account; otherwise an attacker could perform a man-in-the-middle attack just by compromising Alice's email address. To deal with a situation where the user's long-term private key is no longer available (e.g., due to a disk failure), Alpenhorn institutes a lockout policy: if 30 days pass without a legitimate attempt to acquire the user's IBE private key, a PKG allows re-registering that email address with a new long-term signing key, using email verification as described above.

An adversary with access to a user's email account may register that email address before the legitimate user has a chance to do so himself. This poses a risk when a user adds a friend: is that friend's account registered by the friend, or by someone else that compromised their email account? To address this issue, it suffices for a user to

[4]To avoid receiving a separate confirmation email from each PKG, Alice could send a single DKIM-signed email message [23] containing her long-term signing key, which each PKG could independently verify.

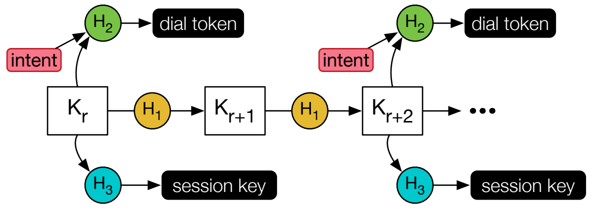


Figure 4: Overview of keywheel operations. K_r is a shared secret key in the keywheel at round r. H_i is a keyed family of cryptographic hash functions (such as HMAC-SHA256), with subscript i denoting the key.

learn one bit of information from their friend: namely, whether they successfully registered for an account in Alpenhorn. This bit can be conveyed informally (e.g., by announcing "contact me using Alpenhorn"), so as to minimize the need to exchange information out-of-band prior to using Alpenhorn. Once a user successfully registers for an account on all PKGs, and connects at least once every 30 days, Alpenhorn's lockout policy ensures that a compromised email account cannot be used to take over the user's Alpenhorn account.

4.7 Computing a shared secret

Once two clients have exchanged add-friend messages through Alpenhorn, they can compute a shared secret using the standard Diffie-Hellman key-exchange protocol. Specifically, the DialingKey in the add-friend message represents the public part of an ephemeral public-private key pair generated by each client for that request. Upon receiving the other party's DialingKey, a client combines its private key with the other party's public key to compute a secret key known only to these two clients. The DialingRound value helps the two clients synchronize their keywheels, as described in the next section.

In summary, Alpenhorn's add-friend protocol allows two clients to establish a shared secret. It achieves privacy for metadata by using distributed IBE with ciphertext anonymity; achieves forward secrecy by using short-lived IBE keys; and achieves authentication through its email registration protocol, PKG signatures on user keys, and optional out-of-band key distribution. The client pseudocode for the add-friend protocol is shown in Algorithm 1.

5 Dialing protocol

Once the add-friend protocol establishes a shared secret between two clients, the dialing protocol allows clients to repeatedly establish conversations and obtain fresh, ephemeral keys for these conversations. The dialing protocol faces two challenges: providing forward secrecy, and providing low latency (compared to the add-friend protocol). The dialing protocol addresses these challenges using a *keywheel* construction, shown in Figure 4. A keywheel stores a shared key, and performs two operations on it.

Algorithm 1 Add-friend round: client

Consider a user Alice, with email address id_{Alice} and signing key pair $(pk_{\text{sign}}^{\text{Alice}}, sk_{\text{sign}}^{\text{Alice}})$. Each mixnet server i $(1 \le i \le n)$ has a short-lived public encryption key pk_{enc}^{i}. Each PKG server j $(1 \le j \le N)$ has a long-term signing key pk_{sign}^{j}, and a short-lived IBE master key pk_{ibe}^{j}. K is the total number of add-friend mailboxes for this round. Alice's client takes the following steps for each round r:

1. **Acquire private keys** (assuming Alice already registered her email address): Alice uses $sk_{\text{sign}}^{\text{Alice}}$ to authenticate to each PKG server. Each server, if authentication succeeds, returns private key $sk_{\text{ibe}}^{j,\text{Alice}}$ and signature σ^j of $(id_{\text{Alice}}, pk_{\text{sign}}^{\text{Alice}}, r)$ using pk_{sign}^{j}.

2a. **Sign and encrypt request** (if Alice wants to introduce herself to Bob, whose email address is id_{Bob}): Create the request $m = (id_{\text{Alice}}, \sum_{j=1}^{N} \sigma^j, pk_{\text{sign}}^{\text{Alice}}, \sigma, pk_{\text{dh}}^{\text{Alice}}, w)$, where $pk_{\text{dh}}^{\text{Alice}}$ is a freshly generated Diffie-Hellman key to be used in dialing round w, and $\sigma = \text{Sign}(sk_{\text{sign}}^{\text{Alice}}, (id_{\text{Alice}}, pk_{\text{dh}}^{\text{Alice}}, w))$. The mailbox is $b = H(id_{\text{Bob}}) \bmod K$. Using IBE, encrypt the request to get $e_{n+1} = (b, \text{Enc}_{\text{IBE}}(\sum_{j=1}^{N} pk_{\text{ibe}}^{j}, id_{\text{Bob}}, m))$.

2b. **Construct fake request** (if Alice does not want to introduce herself to anyone this round): Set $e_{n+1} = (K, 0^{\ell})$ where ℓ is the length of an IBE-encrypted request as above.

3. **Onion wrap the request and send it to the mixnet**: Encryption happens in reverse, from server n to server 1, as server 1 will be the first to decrypt the request. For each server i, generate a temporary keypair (pk_i, sk_i). Then, re-encrypt e_{i+1} with $s_i = \text{DH}(sk_i, pk_{\text{enc}}^{i})$ to get $e_i = (pk_i, \text{Enc}(s_i, e_{i+1}))$.

4. **Download and scan mailbox**: Download the mailbox $H(id_{\text{Alice}}) \bmod K$. For each ciphertext c in the mailbox, attempt $(m, ok) = \text{Dec}_{\text{IBE}}(\sum_{j=1}^{N} sk_{\text{ibe}}^{j,\text{Alice}}, c)$. If decryption succeeds, then $m = (id, \sigma_{\text{servers}}, pk_{\text{sign}}^{id}, \sigma, pk_{\text{dh}}^{id}, w)$.

   Let $ok_1 = \text{Verify}(\sum_{j=1}^{N} pk_{\text{sign}}^{j}, \sigma_{\text{servers}}, (id, pk_{\text{sign}}^{id}, r))$ and let $ok_2 = \text{Verify}(pk, \sigma, (id, pk_{\text{dh}}^{id}, w))$. If $ok_1 \wedge ok_2$, then notify the user of the friend request from id.

5. **Compute shared secret**: If id is a new friend and Alice accepts the request, generate a fresh Diffie-Hellman keypair $(pk_{\text{dh}}^{\text{Alice}}, sk_{\text{dh}}^{\text{Alice}})$, and send an add-friend request with $pk_{\text{dh}}^{\text{Alice}}$ to id in the next round. Otherwise, id is a friend Alice added in a previous round with keypair $(pk_{\text{dh}}^{\text{Alice}}, sk_{\text{dh}}^{\text{Alice}})$, and now the friend is confirmed. In either case, compute shared secret $s = \text{DH}(sk_{\text{dh}}^{\text{Alice}}, pk_{\text{dh}}^{id})$ and add (id, s, w) to the keywheel table.

First, in every round of the dialing protocol, the keywheel updates the key, by hashing it with a cryptographically secure hash function. This is represented by the evolution of k_r into k_{r+1} and so on in Figure 4. By updating the key, a client ensures that an adversary that compromises a client at some time will be unable to obtain any keywheel state from prior rounds. Alpenhorn clients securely erase the old key when performing keywheel updates.

Second, the keywheel can generate *dial tokens* that the user will send out to signal their intention to call a friend. Dial tokens are generated by applying a different hash function to the current round's key. Figure 4 shows the client generating dial tokens in rounds r and $r + 2$. A dial token is a 256-bit value; this is much shorter than the size of the request in the add-friend protocol, and allows the dialing protocol to be efficient. Since the 256-bit dial token is pseudo-random, an adversary that does not know the keywheel state of two friends cannot predict what dial token they might use in a given round. An additional *intent* is hashed along with the key, as we will describe shortly.

Alpenhorn clients call each other by sending dial tokens to a mailbox through the Alpenhorn mixnet. To call a friend, an Alpenhorn client simply computes the dial token for a given round, with an application-supplied intent value, and sends it through the mixnet. To check if a friend is calling, a client downloads the list of dial tokens for that round, and computes all possible dial tokens that each of its friends could have sent in that round. Since two friends have the exact same keywheel state in a given round, a client can easily compute all of the possible incoming dial tokens, by enumerating all of its friends, and all possible intent values; this is cheap to do because hashing is fast and the number of intents is typically small.

If a client finds a dial token from a friend in the mailbox, the client invokes the `IncomingCall` callback to inform the application of the incoming call. The session key for the conversation is computed by hashing that round's key from the keywheel with a different hash function, as shown in Figure 4. The use of this hash function is a precaution, so that if an application inadvertently leaks a session key, it does not compromise future keywheel states.

Finally, Alpenhorn encodes the set of dial tokens in a mailbox using a Bloom filter to reduce the client bandwidth required to download the contents of a mailbox.

5.1 Keywheel synchronization

An Alpenhorn client maintains keywheels for each friend, as shown in Figure 5, consisting of a shared key and a round number. When two friends establish a shared secret through the add-friend protocol, this shared secret is added to the clients' respective keywheel tables. It is important for the clients to agree on the round number corresponding to this initial key; Alpenhorn uses the `DialingRound` field from the add-friend request for this purpose.

To maintain forward secrecy, an Alpenhorn client must update the keywheel state over time and discard old keys. However, the client needs to be able to generate dial tokens for the current round, and to check for incoming dial tokens. Thus, the client advances its keywheels to round $r + 1$ as soon as it has both sent any possible dial requests

Alice's Keywheel table at round 25

Friend	Secret Key	Round
bob@gmail.com	gZbkHyECIhQJ0XaQcKm	25
joanna@foo.edu	s1lJ5kRWp73M4WEMI09	25
chris@hotmail.com	W9uoocTsoYToW1A7nH7	28

$\downarrow$

Alice's Keywheel table at round 26

Friend	Secret Key	Round
bob@gmail.com	AUuJw64TXCAFabdbCGp	26
joanna@foo.edu	z3XukuxRR4dUnkrWpYr	26
chris@hotmail.com	W9uoocTsoYToW1A7nH7	28

Figure 5: Evolution of a client's keywheel table. The keywheel entry for `chris@hotmail.com` has a round number higher than the current round because it was recently established through the add-friend protocol, and Chris's client supplied a `DialingRound` value of 28.

for round *r*, and checked the mailbox from round *r* for possible incoming dial tokens.

If a client cannot download the mailbox for some round, it keeps retrying; the mailbox contents is public state and is maintained by the Alpenhorn servers for a relatively long time. After some time (e.g., a day), the Alpenhorn client gives up trying to fetch the mailbox for an old round, and advances the keywheels to preserve forward secrecy.

5.2 Bloom filter encoding

Alpenhorn optimizes the dialing protocol by encoding the mailbox contents as a Bloom filter [5], which reduces the size of the mailbox that the clients must download, and in turn enables the dialing protocol to run more frequently. The encoding is done by the last server in the mixnet, which is responsible for choosing the optimal parameters to encode a given number of dial tokens into a single Bloom filter.

A Bloom filter allows clients to determine if a dial token is present in the mailbox, with no false negatives, and a small probability of a false positive. No false negatives means that Alpenhorn never misses an incoming call. A false positive translates into the Alpenhorn client invoking the `IncomingCall` callback even though the friend did not initiate a call. Alpenhorn tunes the Bloom filter to provide a low false positive rate of 10^{-10} (which roughly translates into one phantom incoming call in over a decade), using 48 bits per element in the Bloom filter. This is a significant savings over the 256-bit size of the dial token.

5.3 Intents

Alpenhorn's target messaging applications have relatively high overheads associated with setting up and tearing down conversations, and may have limits on how many active connections a user may have at a time. For instance, Vuvuzela allows a user to be active in only one conversation at a time, so if a user receives an incoming call, they may need to drop one conversation to start a different one.

To convey additional information, Alpenhorn allows an application to pass a small integer along with a call, to help the recipient decide how to respond to the incoming call, before a conversation is established. For example, the following might be useful intents for a messaging application to inform the recipient of the nature of the call: (1) let's chat right now; (2) let's chat soon; (3) call me back when you're free. An application informs the Alpenhorn client ahead of time how many intents it plans to use, so that the client can enumerate all possible incoming dial tokens.

6 Sender anonymity

Alpenhorn uses the Vuvuzela mixnet design [41] to ensure that an adversary cannot determine which client sent any given request in a mailbox, and cannot correlate a user's requests with mailbox activity over time (more precisely, Alpenhorn achieves differential privacy, as formalized for a messaging protocol by Vuvuzela).

The mixnet works by arranging a fixed, small number of servers in a chain (e.g., three servers). Each server receives all of the requests for a round, decrypts them using its private key, re-orders them randomly, and sends them to the next server in the mixnet chain. Each server also adds a number of noise messages destined to each mailbox, chosen according to a Laplace distribution with a configurable mean amount of noise μ. As long as one server in the mixnet chain is honest (i.e., does not reveal either its private key or its random re-ordering), an adversary cannot determine which incoming request (if any, due to noise) corresponds to a particular outgoing request.

Achieving good performance requires striking a balance in terms of the number of mailboxes. If there are too few mailboxes, each mailbox will contain a large number of requests, and clients will have to download a lot of data each round. If there are too many mailboxes, the servers will be overwhelmed by noise requests, since *each* mailbox receives the same average amount of noise, regardless of how many mailboxes there are. Alpenhorn aims to strike a good balance by ensuring there is a roughly equal amount of noise and real requests in each mailbox.

7 Implementation

To evaluate Alpenhorn's design, we implemented a prototype in approximately 10,000 lines of Go code:

https://github.com/vuvuzela/alpenhorn

Our implementation of IBE uses the BN-256 elliptic curve [3] (implemented in AMD64 assembly [34]), which targets the 128-bit security level. Recent improvements in cryptanalysis that were published after we built our prototype suggest that BN-256 actually provides less than 96 bits of security [27]. We hope to adopt a more suitable curve in the future to address this, but we do not expect this to have a dramatic impact on our performance results.

Our prototype implements an *entry server*, which is separate from the mixnet and IBE servers. The entry server's job is to manage a large number of WebSocket connections from clients, announce when a new round is starting, and aggregate client requests into a single batch that is sent to the Alpenhorn servers. The entry server is not trusted.

Finally, to distribute the add-friend and dialing mailboxes to many users, our prototype relies on a content distribution network (CDN), such as Akamai.

8 Evaluation

We quantitatively answer the following questions:

- What is the latency for adding a friend and initiating a conversation through Alpenhorn, and what is the client overhead imposed by Alpenhorn? (§8.2)
- Can Alpenhorn support a large number of users, and how does it scale when adding more servers? (§8.3)
- How does Alpenhorn handle skewed workloads, where some users are highly popular? (§8.4)
- How much effort is required to integrate Alpenhorn into a private messaging application? (§8.5)
- How would Alpenhorn's performance be impacted if new weaknesses are discovered in the pairing-based cryptography that Alpenhorn's IBE relies on? (§8.6)

The results suggest that Alpenhorn can provide acceptable performance for private text messaging applications that tolerate latency on the order of minutes, such as Vuvuzela.

8.1 Experimental setup

To answer some of the above questions, we ran experiments on Amazon EC2. Each server ran on a `c4.8xlarge` virtual machine running Linux 4.4 with 36 Intel Xeon E5-2666 v3 CPU cores, 60 GB of RAM, and 10 Gbps of network bandwidth. We compiled the code with Go 1.7.

Unless otherwise specified, our experiments used a chain of three servers, each corresponding to one VM. Each of these servers also ran a PKG. We used one additional VM to run the entry server. The first server in the chain and the entry server were located in the Virginia EC2 region. The second server was located in Ireland and the third server in Frankfurt, Germany. For experiments with more servers, we used the same three regions in a cycle.

Clients were simulated on five `c4.8xlarge` VMs in Virginia (each individual client was limited to using at most 4 cores). To avoid establishing millions of TCP connections, we opened 1,000 connections from each client VM to the entry server, and assigned multiple clients to each TCP connection. We did not use a CDN in our experiments for distributing mailboxes; instead, only a small number of clients downloaded their mailbox in each round (enough to report sound measurements). Each round, 5% of generated requests were real (not cover traffic). For example, to simulate one million users in the add-friend protocol, we generated 50,000 `AddFriend` requests, and 950,000 cover traffic messages. For dialing experiments, each client had 1,000 friends in their address book, and the maximum number of intents was 10. Unless otherwise noted, our experiments assume that all users are equally popular.

Each mixnet server adds an average of $\mu = 4,000$ noise messages to each add-friend mailbox and $\mu = 25,000$ noise messages to each dialing mailbox. With Laplace b parameters of 406 and 2,183 respectively, each protocol achieves ($\varepsilon = \ln 2$, $\delta = 10^{-4}$)-differential privacy for 900 add-friend requests and 26,000 calls (e.g., 7 calls per day for 10 years). For our experiments, we set $b = 0$ to reduce the variance in the results. The Vuvuzela paper discusses the implications of differential privacy parameters in detail.

8.2 Client performance

Deploying Alpenhorn requires the application developer to decide how frequently to run the add-friend and dialing rounds. The main consideration is a trade-off between latency and client bandwidth: more frequent rounds reduce latency but require clients to download mailboxes more frequently (the rest of this section quantifies this trade-off). We expect that Alpenhorn would be used in settings where users do not expect an instant response to friend requests (similar to adding a friend on Facebook), and where users do not add new friends very often. In this setting, the latency of the dialing protocol is more important than the add-friend latency, since the add-friend protocol needs to happen only once between pairs of users, and all further attempts to communicate use the dialing protocol.

Bandwidth. We compute the latency and bandwidth requirements of add-friend and dialing. The crucial parameters that affect latency and bandwidth are round duration (how long Alpenhorn waits to process the next batch of requests for the next round), and the number of active users, which affects the number of requests processed in a round.

The duration of add-friend and dialing rounds are parameters to the system that can be used to adjust client bandwidth usage. Figure 6 shows the total client-side bandwidth requirement of the add-friend protocol as a function of the round duration. The bandwidth is mostly spent on downloading add-friend mailboxes. In Figure 6, with one million users, each add-friend mailbox contains around 12,000 friend requests from users and around 12,000 friend requests from noise (4,000 per server, on average), for a total of 24,000 requests. At 308 bytes per request, the add-friend mailbox is around 7.4 MB every round. As described in §6, when the number of requests goes up, the mixnet increases the number of mailboxes,

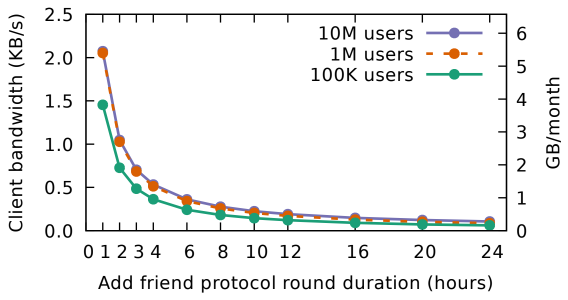


Figure 6: Required client-side bandwidth for the add-friend protocol when varying the round duration.

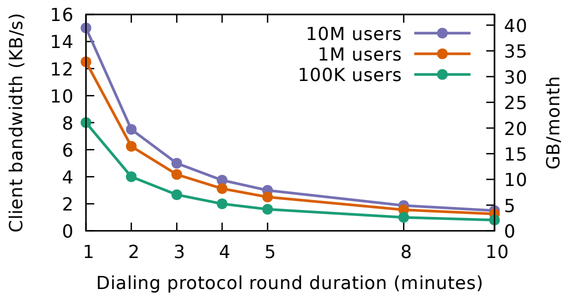


Figure 7: Required client-side bandwidth for the dialing protocol when varying the round duration.

thus ensuring that the size of the mailbox stays roughly constant. (With 100K users, the number of messages each round is less than 12,000, so the single mailbox is smaller than 7.4 MB.)

The dialing protocol analysis is shown in Figure 7. Nearly all of the client bandwidth is spent on downloading the Bloom filter that is scanned for dial tokens. With 1M users and 5% active, Alpenhorn uses one Bloom filter to encode the 125,000 received dial tokens. At 48 bits per token, the Bloom filter is 0.75 MB each round. With 10 million users and 500K active, Alpenhorn distributes the incoming dial tokens into 7 distinct Bloom filters (mailboxes). Each Bloom filter has a roughly equal amount of noise and user data (75,000 each), so the total Bloom filter size is 0.9 MB per user per round. If Alpenhorn uses a dialing round duration of 5 minutes, then the average client bandwidth is 3 KB/s, or 7.8 GB per month (manageable for a cellphone with occasional WiFi connectivity). With a round duration of 5 minutes, the average end-to-end latency for `Call` requests is about 2.5 minutes.

CPU. We measured the client-side CPU usage of both protocols. Our implementation of IBE can compute 800 decryptions per second per core. Thus, to scan a mailbox with 24,000 friend requests takes 8 seconds on a 4 core machine. The CPU cost of dialing is tiny in comparison. One core can compute 1 million hashes per second. Even if a user has 1,000 friends in their keywheel (much more than the average number of friends on Facebook [21, 40]), and the application uses 10 intent values, the Bloom filter can be scanned in less than a second.

Key extraction. We measured the time it takes a client to obtain the combined identity private key for a single round from the PKGs, with a varying number of PKGs running in the same EC2 region as the client. With 3 PKGs, the median latency was 4.9 msec (ranging from 4.7 msec to 7.6 msec) over 100 runs. With 10 PKGs, the median latency was 5.2 msec (ranging from 4.7 msec to 10.8 msec). This suggests that, for a client, there is almost no cost to additional PKGs aside from the network latency of contacting the servers.

8.3 Server performance

To evaluate whether Alpenhorn can support a large number of users, we measured the time it takes for the servers to complete a round, varying the number of users and servers. Specifically, we measured the latency of `AddFriend` and `Call` requests, assuming the client sends the requests at the optimal time just before the round closes, and measuring time until the client downloads the mailbox and finishes scanning all requests. Thus, our latency measurements do not include artificial delays imposed by the servers to reduce bandwidth costs (in an actual deployment, servers would be idle most of the time, because the interval at which new rounds start is much higher than the time it takes to complete a round, to keep client bandwidth reasonable). We also measured the throughput of PKG servers generating users' IBE private keys.

Add-friend. Figure 8 shows the median, minimum, and maximum observed latencies as we varied the number of users, for different numbers of servers. With 10 million users, the median 3-server round latency is 152 seconds. Adding more servers increases the latency due to the additional processing that each server must perform, and due to the additional noise introduced by additional servers.

Dialing. Figure 9 shows the latency for dialing as we varied the number of users. The graph shows that Alpenhorn can support 10 million users on 3 servers with a latency of 118 seconds. The latency increases with more servers much as with the add-friend protocol.

PKG servers. The PKG servers in Alpenhorn have to extract a private key for every user in every round, which can place a lower bound on how frequently add-friend rounds can run. In our experiments, a PKG takes 232 seconds to respond to 1 million user key extraction requests (4310 requests per second). This suggests that, even with 10 million users, each PKG can extract the keys of all users in well under an hour.

8.4 Skewed popularity

To evaluate Alpenhorn's performance under a skewed workload, we measured the latency of `AddFriend` and `Call` rounds, as above, with a varying user popularity

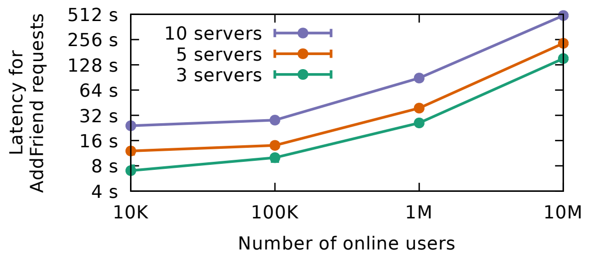


Figure 8: Performance of Alpenhorn's add-friend protocol when varying the number of users online.

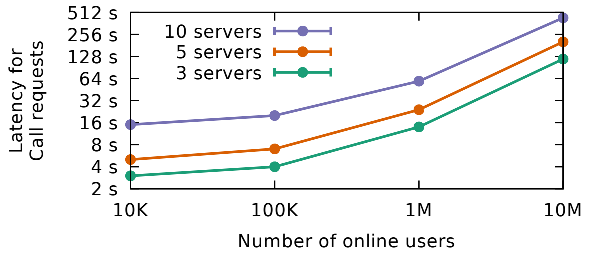


Figure 9: Performance of Alpenhorn's dialing protocol when varying the number of users online.

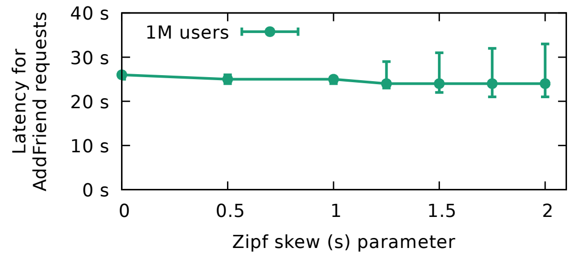


Figure 10: Latency for Alpenhorn's add-friend rounds when varying the skew of the user popularity.

distribution. In particular, instead of choosing the recipient of `AddFriend` or `Call` uniformly at random, in this experiment the recipient is chosen according to a Zipf distribution; that is, the probability of picking some user i, from 1 to N (the number of users) is proportional to i^{-s}.

Figure 10 show the results for the add-friend protocol for 1M users and 3 servers. The median latency stays constant even as user popularity becomes highly skewed (e.g., at $s = 2$, the top 10 users receive 94.2% of all requests). However, as the skew increases, the maximum latency increases (and the minimum decreases) because some mailboxes contain more messages (if they happen to correspond to highly popular users), and other mailboxes become smaller. Even for highly skewed distributions, the effect is not dramatic because Alpenhorn mailboxes already contain a significant amount of noise (about half of the messages, on average) regardless of where users choose to send messages that round. With 1M users and $s = 2$, the largest mailbox is 14.95 MB and the smallest is 4.15 MB.

Dialing is less affected by skew because the client CPU time to scan a mailbox is negligible. With 10 million users and $s = 2$, the minimum and maximum latencies are 119 and 120 seconds respectively, and the minimum and maximum mailbox sizes are 231 KB and 1.39 MB.

8.5 Application integration

To evaluate whether Alpenhorn fits with private messaging applications, we integrated it with Vuvuzela and Pond.

Vuvuzela had its own dialing protocol for starting a conversation (which assumed out-of-band public key distribution and did not provide forward secrecy), which we replaced entirely with Alpenhorn. We had to change 200 lines of code; this does not include deleting all of the code from the old dialing protocol. We had to tweak the Vuvuzela conversation protocol, since it expected a public key as input, rather than a shared secret (as provided by `Call`). Our changes also added two new commands to the Vuvuzela client, `/addfriend` and `/call`, which tie directly into the Alpenhorn API. All other Vuvuzela components remain unchanged. The resulting Vuvuzela client that uses Alpenhorn provides the same security guarantees as Vuvuzela (including differential privacy), with the addition of forward secrecy for bootstrapping conversations (which Vuvuzela's original dialing protocol did not provide).

Pond also provides its own bootstrapping protocol called PANDA [2]. PANDA assumes that pairs of users have a shared secret, and provides a GUI for entering that secret. We built a standalone Alpenhorn client that lets users friend and call each other from a basic command-line interface. The client prints the resulting shared secret to the screen, which the users can then paste into PANDA. This eliminates the need to generate a shared secret out-of-band.

8.6 Cryptographic strength

In light of recent attacks on the BN-256 curve [27], which Alpenhorn uses for IBE, it may be necessary to switch to a different curve or IBE construction to maintain Alpenhorn's security guarantees in the future. Since we cannot predict what scheme may provide the best alternative in the future, this section analytically evaluates the impact of such a switch on Alpenhorn's performance.

The IBE construction impacts three aspects of Alpenhorn's performance: CPU cost on the PKG for generating identity private keys, CPU cost on the client for decrypting the add-friend mailbox, and bandwidth for downloading the add-friend mailbox. PKG and client CPU costs would be directly proportional to the respective CPU costs of any new IBE construction. The bandwidth impact, on the other hand, is a bit more subtle. Alpenhorn's current add-friend request is 244 bytes plus the size of an IBE ciphertext (encrypting a symmetric key that encrypts the rest of the request); the IBE ciphertext is 64 bytes in our prototype. This suggests that changes to the curve or IBE scheme used by Alpenhorn should result in linear or sub-linear impacts on Alpenhorn's performance.

9 Discussion and Limitations

Client compromise. If an adversary compromises an Alpenhorn client (i.e., obtains the user's long-term signing private key and the user's keywheel state), the user must generate a new long-term signing key and new keywheels to re-establish security, as we now discuss.

Registering the new long-term signing key faces two complications. First, the adversary can keep using the stolen signing key, thereby preventing the user from re-registering the same email address (since the PKGs implement a 30-day lockout policy). To address this problem, the user should issue a *deregister* command to the PKGs signed by their old key. The second issue is that, once an account is deregistered, an adversary may be able to register his own key under the user's email address, since they likely got access to the user's email account when they compromised the user's machine. We address this issue by placing the account into a 30-day lockout period after deregistration. This way, if the user can re-establish access to their email account within 30 days of the compromise (e.g., through out-of-band authentication with the email provider), they can regain their Alpenhorn account.

Establishing new keywheels with friends requires the user to simply re-run the add-friend protocol with each friend. To guard against the possibility of an adversary corrupting the list of friend long-term signing keys stored on the user's computer, we recommend that the user keep an offline backup of long-term signing keys of friends, and restore from backup to recover from a compromise. On the other hand, we discourage users from backing up their keywheel, since that is bad for forward secrecy.

Lost client state. If the state of an Alpenhorn client is lost (e.g., because the user physically lost their laptop), the user should follow the steps described above for recovering from a compromised client. The only difference is that the user no longer has access to the long-term signing private key, so the user cannot explicitly deregister. However, the user can simply wait for the same lockout period until re-registering their account through email validation.

DoS attacks. A malicious group of users might attempt to cause a denial of service attack by sending friend or dialing requests in every round (rather than cover traffic) in order to fill mailboxes. This can in turn increase client bandwidth, and, since Alpenhorn will create additional mailboxes to compensate for the extra load, cause the mixnet servers to incur a higher CPU cost to generate noise for the extra mailboxes. To address this, Alpenhorn servers could issue a limited number of blinded signatures to each user every day, and reject any requests that don't have a valid unblinded signature. Since the signatures are blinded, this approach would not leak metadata.

Users going offline. Alpenhorn does *not* assume that users stay online all the time (Alpenhorn avoids intersection attacks [32] by using constant-rate cover traffic to and from all client machines and by using noise to ensure differential privacy of observable mailboxes). However, Alpenhorn does assume that the user's observable activity, which includes going online and offline, is not highly correlated with any confidential metadata they want to keep private. A straightforward way to achieve this is to keep Alpenhorn running all the time, but this may not be practical for users with mobile devices.

An example scenario that illustrates the above problem would be two users that both close their laptops at the same time after finishing a conversation; an adversary that observes both of them going offline at the same time may infer that both of them could have been talking just before. One approach to address this that we hope to explore in future work is to require the users to stay online for a random length of time after finishing a conversation, and to use differential privacy to precisely reason about what random time intervals would be required.

10 Conclusion

Alpenhorn is the first system for establishing session keys between pairs of users that does not require out-of-band communication aside from knowing a user's Alpenhorn username (their email address), and that provides privacy and forward secrecy for metadata, assuming that at least one server is uncompromised. Alpenhorn achieves this by using identity-based encryption (IBE) in a novel way to determine another user's public key without revealing metadata in an anytrust setting. Alpenhorn ensures forward secrecy for all data by refreshing IBE keys and by storing client-side secrets in a *keywheel*. The keywheel provides a bandwidth-efficient means for calling existing friends and starting conversations. Together, these techniques enable Alpenhorn to bootstrap communication in messaging systems that support 10 million users.

Acknowledgments

Thanks to Tej Chajed, Chucky Ellison, Jon Gjengset, Jelle van den Hooff, Frans Kaashoek, and Malte Schwarzkopf for helping improve this paper, and especially thanks to Vinod Vaikuntanathan for helping us prove the security of Anytrust-IBE. Thanks also to the anonymous reviewers. This work was supported by NSF awards CNS-1053143 and CNS-1413920, and by Google.

References

[1] S. Angel and S. Setty. Unobservable communication over fully untrusted infrastructure. In *Proceedings of the 12th Symposium on Operating Systems Design and Implementation (OSDI)*, Savannah, GA, Nov. 2016.

[2] J. Appelbaum et al. Going dark: Phrase automated nym discovery authentication, 2013. https://github.com/agl/pond/tree/master/papers/panda.

[3] P. S. L. M. Barreto and M. Naehrig. Pairing-friendly elliptic curves of prime order. In *Selected Areas in Cryptography – SAC 2005*, volume 3897 of *Lecture Notes in Computer Science*, pages 319–331, 2006.

[4] E. Ben-Sasson, A. Chiesa, C. Garman, M. Green, I. Miers, E. Tromer, and M. Virza. Zerocash: Decentralized anonymous payments from Bitcoin. In *Proceedings of the 35th IEEE Symposium on Security and Privacy*, San Jose, CA, May 2014.

[5] B. H. Bloom. Space/time trade-offs in hash coding with allowable errors. *Communications of the ACM*, 13(7):422–426, 1970.

[6] A. Boldyreva, V. Goyal, and V. Kumar. Identity-based encryption with efficient revocation. In *Proceedings of the 15th ACM Conference on Computer and Communications Security (CCS)*, pages 417–426, Alexandria, VA, Oct. 2008.

[7] D. Boneh and M. K. Franklin. Identity-based encryption from the Weil pairing. In *Proceedings of the 21st Annual International Cryptology Conference (CRYPTO)*, Santa Barbara, CA, Aug. 2001.

[8] D. Boneh, B. Lynn, and H. Shacham. Short signatures from the Weil pairing. *Journal of Cryptology*, 17(4):297–319, 2004.

[9] N. Borisov, I. Goldberg, and E. Brewer. Off-the-record communication, or, why not to use PGP. In *Proceedings of the 2004 Workshop on Privacy in the Electronic Society*, Washington, DC, Oct. 2004.

[10] N. Borisov, G. Danezis, and I. Goldberg. DP5: A private presence service. In *Proceedings of the 15th Privacy Enhancing Technologies Symposium*, Philadelphia, PA, June–July 2015.

[11] X. Boyen. Multipurpose identity-based signcryption: A Swiss army knife for identity-based cryptography. In *Proceedings of the 23rd Annual International Cryptology Conference (CRYPTO)*, Santa Barbara, CA, Aug. 2003.

[12] X. Boyen and B. Waters. Anonymous hierarchical identity-based encryption (without random oracles). Cryptology ePrint Archive, Report 2006/085, June 2006.

[13] J. Brooks et al. Ricochet: Anonymous instant messaging for real privacy, 2016. https://ricochet.im.

[14] R. Canetti, S. Halevi, and J. Katz. A forward-secure public-key encryption scheme. *Journal of Cryptology*, 20(3):265–294, July 2007.

[15] D. Chaum, F. Javani, A. Kate, A. Krasnova, J. de Ruiter, A. T. Sherman, and D. Das. cMix: Anonymization by high-performance scalable mixing. Cryptology ePrint Archive, Report 2016/008, Jan. 2016.

[16] L. Chen, Z. Cheng, and N. P. Smart. Identity-based key agreement protocols from pairings. http://eprint.iacr.org/, June 2006.

[17] C. Cocks. An identity based encryption scheme based on quadratic residues. In *Proceedings of the 8th Proceedings of the 8th IMA International Conference on Cryptography and Coding*, Cirencester, UK, Dec. 2001.

[18] C. Conley. *Metadata: Piecing together a privacy solution*. ACLU of California, Feb. 2014. https://www.aclunc.org/publications/metadata-piecing-together-privacy-solution.

[19] H. Corrigan-Gibbs, D. Boneh, and D. Mazières. Riposte: An anonymous messaging system handling millions of users. In *Proceedings of the 36th IEEE Symposium on Security and Privacy*, San Jose, CA, May 2015.

[20] R. Dingledine, N. Mathewson, and P. Syverson. Tor: The second-generation onion router. In *Proceedings of the 13th Usenix Security Symposium*, pages 303–320, San Diego, CA, Aug. 2004.

[21] Edison Research. Average number of Facebook friends of users in the United States as of February 2014, by age group. Statista - The Statistics Portal, Mar. 2014. https://www.statista.com/statistics/232499/americans-who-use-social-networking-sites-several-times-per-day/.

[22] J. A. Halderman, S. D. Schoen, N. Heninger, W. Clarkson, W. Paul, J. A. Calandrino, A. J. Feldman, J. Appelbaum, and E. W. Felten. Lest we remember: Cold boot attacks on encryption keys. In *Proceedings of the 17th Usenix Security Symposium*, San Jose, CA, July–Aug. 2008.

[23] T. Hansen, D. Crocker, and P. Hallam-Baker. DomainKeys Identified Mail (DKIM) service overview. RFC 5585, Network Working Group, July 2009.

[24] A. Iliev and S. Smith. Privacy-enhanced credential services. In *Proceedings of the 2nd Annual NIST PKI Research Workshop*, Apr. 2003.

[25] A. Kate and I. Goldberg. Distributed private-key generators for identity-based cryptography. In *Proceedings of the 7th Conference on Security and Cryptography for Networks*, Sept. 2010.

[26] Keybase. Keybase, 2016. https://keybase.io/.

[27] T. Kim and R. Barbulescu. Extended tower number field sieve: A new complexity for the medium prime case. In *Proceedings of the 36th Annual International Cryptology Conference (CRYPTO)*, Santa Barbara, CA, Aug. 2016.

[28] A. Kwon, D. Lazar, S. Devadas, and B. Ford. Riffle: An efficient communication system with strong anonymity. In *Proceedings of the 16th Privacy Enhancing Technologies Symposium*, Darmstadt, Germany, July 2016.

[29] A. Langley. Pond, 2016. https://github.com/agl/pond.

[30] D. Lazar and N. Zeldovich. Alpenhorn: Bootstrapping secure communication without leaking metadata (extended technical report). Technical report, MIT Computer Science and Artificial Intelligence Laboratory, Cambridge, MA, Oct. 2016. Also available at https://vuvuzela.io/alpenhorn-extended.pdf.

[31] W. Lueks and I. Goldberg. Sublinear scaling for multi-client private information retrieval. In *Proceedings of the 19th International Conference on Financial Cryptography and Data Security*, Jan. 2015.

[32] N. Mathewson and R. Dingledine. Practical traffic analysis: Extending and resisting statistical disclosure. In *Proceedings of the Privacy Enhancing Technologies Workshop*, pages 17–34, May 2004.

[33] J. Mayer, P. Mutchler, and J. C. Mitchell. Evaluating the privacy properties of telephone metadata. *Proceedings of the National Academy of Sciences (PNAS)*, 113(20):5536–5541, 2016.

[34] M. Naehrig, R. Niederhagen, and P. Schwabe. New software speed records for cryptographic pairings. In *Progress in Cryptology – LATINCRYPT 2010*, volume 6212 of *Lecture Notes in Computer Science*, pages 109–123, 2010.

[35] Namecoin. Namecoin, 2016. https://namecoin.info/.

[36] T. Perrin and M. Marlinspike. Double ratchet algorithm, 2016. https://github.com/trevp/double_ratchet/wiki.

[37] A. Piotrowska, J. Hayes, N. Gelernter, G. Danezis, and A. Herzberg. AnoNotify: A private notification service. Cryptology ePrint Archive, Report 2016/466, May 2016.

[38] A. Rusbridger. The Snowden leaks and the public. *The New York Review of Books*, Nov. 2013.

[39] A. Shamir. Identity-based cryptosystems and signature schemes. In *Proceedings of the 4th Annual International Cryptology Conference (CRYPTO)*, Santa Barbara, CA, Aug. 1984.

[40] J. Ugander, B. Karrer, L. Backstrom, and C. Marlow. The anatomy of the Facebook social graph. *CoRR*, abs/1111.4503, Nov. 2011. URL http://arxiv.org/abs/1111.4503.

[41] J. van den Hooff, D. Lazar, M. Zaharia, and N. Zeldovich. Vuvuzela: Scalable private messaging resistant to traffic analysis. In *Proceedings of the 25th ACM Symposium on Operating Systems Principles (SOSP)*, Monterey, CA, Oct. 2015.

[42] WhatsApp. WhatsApp encryption overview, Apr. 2016. https://www.whatsapp.com/security/WhatsApp-Security-Whitepaper.pdf.

[43] D. I. Wolinsky, H. Corrigan-Gibbs, B. Ford, and A. Johnson. Dissent in numbers: Making strong anonymity scale. In *Proceedings of the 10th Symposium on Operating Systems Design and Implementation (OSDI)*, Hollywood, CA, Oct. 2012.

Big Data Analytics over Encrypted Datasets with Seabed

Antonis Papadimitriou[1†], Ranjita Bhagwan[*], Nishanth Chandran[*], Ramachandran Ramjee[*],
Andreas Haeberlen[†], Harmeet Singh[*], Abhishek Modi[*], Saikrishna Badrinarayanan[1‡]
[†]University of Pennsylvania, [*]Microsoft Research India, [‡]UCLA

Abstract

Today, enterprises collect large amounts of data and leverage the cloud to perform analytics over this data. Since the data is often sensitive, enterprises would prefer to keep it confidential and to hide it even from the cloud operator. Systems such as CryptDB and Monomi can accomplish this by operating mostly on encrypted data; however, these systems rely on expensive cryptographic techniques that limit performance in true "big data" scenarios that involve terabytes of data or more.

This paper presents Seabed, a system that enables efficient analytics over large encrypted datasets. In contrast to previous systems, which rely on *asymmetric* encryption schemes, Seabed uses a novel, additively *symmetric* homomorphic encryption scheme (ASHE) to perform large-scale aggregations efficiently. Additionally, Seabed introduces a novel randomized encryption scheme called *Splayed ASHE*, or SPLASHE, that can, in certain cases, prevent frequency attacks based on auxiliary data.

1 Introduction

Consider a retail business that has customer and sales records from various store locations across the world. The business may be interested in analyzing these records – perhaps to better understand how revenue is growing in various geographic locations, or which demographic segments of the population its customers are coming from. To answer these questions, the business might rely on a Business Intelligence (BI) system, such as PowerBI [5], Tableau [7], or Watson Analytics [8]. These systems can scale to large data sets, and their turnaround times are low enough to answer interactive queries from customers. Internally, they rely on the cloud to provide the necessary resources at relatively low cost.

However, storing sensitive business data on the cloud can raise privacy concerns, which is why many enterprises are reluctant to use cloud-based analytics solutions. These concerns could be mitigated by keeping the data in the cloud encrypted, so that a data leak (e.g., due to a hacker attack or a rogue administrator) would cause little or no damage. Systems like CryptDB [37] and Monomi [41] can accomplish this by using a mix of different encryption schemes, including deterministic encryption schemes [12] and partially homomorphic cryptosystems; this allows certain computations to be performed directly on encrypted data. However, this approach has two important drawbacks. First, these cryptosystems have a high computational cost. This cost is low enough to allow interactive queries on medium-size data sets with perhaps tens of gigabytes, but many businesses today collect terabytes of data [13, 29, 30, 40]. Our experimental results show that, at this scale, even on a cluster with 100 cores, it would take hundreds of seconds to process relatively simple queries, which is too slow for interactive use. Second, deterministic encryption is vulnerable to frequency attacks [33], which can cause some data leakage despite the use of encryption.

This paper makes two contributions towards addressing these concerns. First, we observe that existing solutions typically use *asymmetric* homomorphic encryption schemes, such as Paillier [35]. This is useful in scenarios where the data is produced and analyzed by different parties: Alice can encrypt the data with the public key and upload it to the cloud, and Bob can then submit queries and decrypt the results with the private key. However, in the case of business data, the data producer and the analyst typically have a trust relationship – for instance, they may be employees of the same business. In this scenario, it is sufficient to use *symmetric* encryption, which is much faster. To exploit this, we construct a new *additively symmetric homomorphic encryption* scheme (or, briefly, ASHE), which is up to three orders of magnitude more efficient than Paillier.

Our second contribution is a defense against frequency attacks based on auxiliary information – a type of attack that has recently been demonstrated in the context of deterministic encryption [33]. For instance, suppose the data contains a column, such as gender, that can take only a few discrete values and that has been encrypted deterministically. If the attacker knows which gender occurs more frequently in the data, she can trivially decode this column based on which ciphertext is the most com-

[1]Part of this work was done while Papadimitriou and Badrinarayanan were doing internships at Microsoft Research India.

mon. We introduce an encryption scheme called *Splayed ASHE (SPLASHE)*, that protects against such attacks by splaying sensitive columns to multiple columns, where each new column corresponds to data for each unique element in the original column. For columns with larger cardinality, SPLASHE uses a combination of splaying and deterministic encryption padded with spurious entries to defeat frequency attacks while still limiting the storage and computational overhead.

We also present a complete system called *Seabed* that uses ASHE and SPLASHE to provide efficient analytics over large encrypted datasets. Following the design pattern in earlier systems, Seabed consists of a client-side planner and a proxy. The planner is applied once to each new data set; it transforms the plain-text schema into an encrypted schema, and it chooses suitable encryption schemes for each column, based on the kinds of queries that the user wants to perform. The proxy transparently rewrites queries for the encrypted schema, it decrypts results that arrive from the cloud, and it performs any computations that cannot be performed directly on the cloud. Seabed contains a number of optimizations that keep the storage, bandwidth, and computation costs of ASHE low, and that make it amenable to the hardware acceleration that is available on modern CPUs.

We have built a Seabed prototype based on Apache Spark [2]. We report results from an experimental evaluation that includes running both AmpLab's Big Data Benchmark [3] and a real, advertising-based analytics application on the Azure cloud. Our results show that, compared to no encryption, Seabed increases the query latency by only 8% to 45%; in contrast, state-of-the-art solutions that are based on Paillier (such as Monomi [41]) would cause an increase by one to two orders of magnitude in query latency.

To summarize, we make the following four contributions in this paper:

- ASHE, an additive symmetric homomorphic encryption scheme that is three orders of magnitude faster than Paillier (Section 3.1);
- SPLASHE, an encryption scheme that protects against frequency-based attacks for fields that require deterministic encryption (Sections 3.3+3.4);
- Seabed, a system that supports efficient analytics over large-scale encrypted data sets (Section 4); and
- a prototype implementation and experimental evaluation of Seabed (Section 6).

2 Overview

Figure 1 shows the scenario we are considering in this paper. A *data collector* gathers a large amount of data, encrypts it, and uploads it to an untrusted cloud platform.

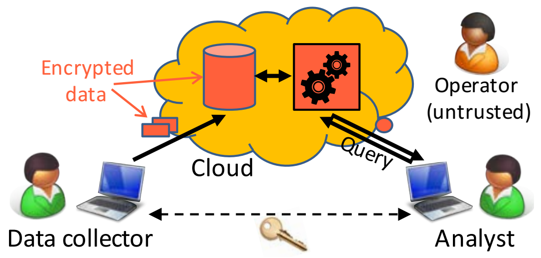


Figure 1: Motivating scenario.

An *analyst* can issue queries to a query processor on the cloud. The responses will be encrypted, but the analyst can decrypt them with a secret key she shares with the data collector.

The workload we wish to support consists of OLAP-style queries on big data sets. As our analysis in Section 5 will show, these queries mostly rely on just a few simple operations (sum, sum-of-squares, etc.), so we focus on these in our server-side design. Our goal is to answer typical BI queries on large data sets within a few seconds – that is, quickly enough for interactive analysis.

2.1 Background

One common approach to solving the above problem is to use *homomorphic encryption*. For instance, there are cryptosystems with an additive homomorphism, such as Paillier [35], which means that it is possible to "add" two ciphertexts $C(x)$ and $C(y)$ to obtain a ciphertext $C(x+y)$ that decrypts to the sum of the two encrypted values. This feature allows the cloud to perform aggregations directly on the encrypted data. There are other systems with different homomorphisms, and even *fully homomorphic* systems [23] that can be used to compute arbitrary functions on encrypted data (Section 7).

Homomorphic encryption schemes are typically randomized, that is, there are many different possible ciphertexts for each value. These schemes enjoy standard semantic (or CPA) security, which informally means that no adversary can learn any information about the plaintext, even given the ciphertext.

However, there are situations where it is useful to let the cloud see some property of the encrypted values (property-preserving encryption). For instance, to compute a join, the cloud *needs* to be able to match up encrypted values, which randomization would prevent. In this case, one can use *deterministic encryption* [12], where each value v is mapped to exactly one ciphertext $C(v)$. However, such schemes are susceptible to *frequency attacks* [33]: if a column can only take a small number of values (say, country), and the cloud knows that some value (say, Canada) will be the most common in the data, it can look for the most common ciphertext and infer that this ciphertext must decrypt to that value. Another example of an operation achievable by a property-preserving encryption scheme is selecting rows based on a range of values (say, timestamps) in an encrypted column. Here, one can use an *order-preserving encryption (OPE)* [15], which can be used to decide whether $x < y$, given only $C(x)$ and $C(y)$. Obviously, if the cloud can perform the comparison, then so can the adversary, so in these schemes, there is a tradeoff between confidentiality, performance, and functionality.

2.2 Threat Model

In this paper, we resolve the above tradeoff in favor of confidentiality and performance. We assume an adversary who is honest but curious (HbC), that is, the adversary will try to learn facts about the data but will not actively corrupt data or otherwise interfere with the system. We do, however, assume that the adversary will attempt to perform frequency attacks as discussed above; this is motivated by recent work [33], and it is the reason we developed SPLASHE.

We are aware that there are much stronger threat models that would prevent the adversary from learning anything at all about the data. However, current solutions for these models, such as using oblivious RAM [34, 26] and fully homomorphic encryption, tend to have an enormous runtime cost (fully homomorphic encryption [23] causes a slowdown by nine orders of magnitude [24]). Our goal is to provide a practical alternative to today's plaintext-based systems (which offer very little security), and this requires keeping the runtime overhead low.

2.3 Alternative approaches

As discussed in Section 2.1, one possible approach to this problem is to use homomorphic encryption. This approach is taken by systems like CryptDB [37] and Monomi [41], which use Paillier as an additive homomorphic scheme. While Paillier is *much* faster than fully homomorphic encryption, it is still expensive. For example, a single addition in Paillier on modern hardware takes about 4 μs (Section 4), so the latency for operations on billions of rows can easily reach several minutes.

An alternative approach is to rely on trusted hardware, such as Intel's SGX [32] or ARM's TrustZone [10]. This approach has a much lower computational overhead, but it introduces new trust assumptions that may not be suitable for all scenarios [19, 20]. It would be good to have options available that offer a low overhead without relying on trusted hardware.

2.4 Our approach

In Seabed, we solve this problem by replacing Paillier with a specially designed *additively symmetric homomorphic encryption* (ASHE) scheme. Since symmetric encryption schemes tend to be much more efficient than asymmetric schemes, this yields a big performance boost (Section 4). Symmetric encryption imposes a restriction that the encrypted data can only be uploaded by someone who has the secret key but this is not a constraint for the typical BI scenario. Thus, the additional protections of asymmetric cryptography are actually superfluous, and the performance gain is essentially "free".

Additionally, in order to protect against frequency attacks that occur when using deterministic or order preserving encryption, we construct a randomized encryption scheme – *SPLayed ASHE*, or SPLASHE that can still enable us to perform many queries on encrypted data that in prior work required deterministic encryption, but without leaking any information on frequency counts. Finally, for those queries that SPLASHE cannot support (e.g., joins), we support deterministic and OPE schemes that leak (a small amount of) information about the underlying plaintext values; we take this decision with the performance of the system in mind.

3 Seabed Encryption Schemes

In this section, we describe the ASHE and SPLASHE schemes in more detail. ASHE and the basic variant of SPLASHE satisfy the standard notion of semantic security (IND-CPA, that leaks no information about plaintext values) while the enhanced variant of SPLASHE provably leaks no more information than the number of dimension values that occur frequently and infrequently in the database. A formal security proof is available in our technical report [36].

3.1 ASHE

ASHE assumes that plaintexts are from the additive group $\mathbb{Z}_n := \{0, 1, \ldots, n-1\}$. It also assumes that the entities encrypting and decrypting a ciphertext (the sender and the recipient, respectively) share a secret key k, as well as a pseudo-random function (PRF) $F_k : I \to \mathbb{Z}_n$ that takes an identifier from a set I and returns a random number from $\mathbb{Z}_n$.

One possible choice for the PRF is $F_k := H(i \,||\, k) \bmod n$ for $i \in I$, where H is a cryptographic hash function (when modeled as a random function), $||$ denotes concatenation and the size of the range of H is a multiple of n. Another choice is AES, when used as a pseudo-random permutation.

Suppose Alice wants to send a value $m \in \mathbb{Z}_n$ to Bob. Then Alice can pick an arbitrary, unique, number $i \in I$ – which we call the *identifier* – and encrypt the message

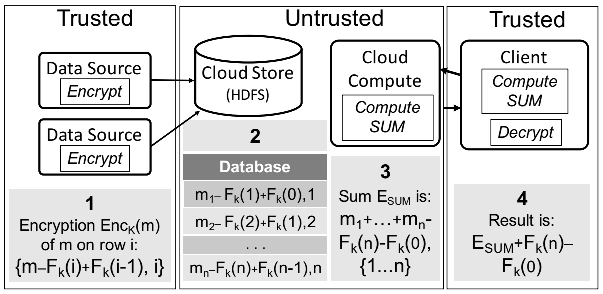


Figure 2: Seabed components and the ASHE scheme.

Plaintext Schema

country	salary
Male	1000
Female	2000
Female	200

Encrypted Schema

sender	salary
DET(Male)	ASHE(1000)
DET(Female)	ASHE(2000)
DET(Female)	ASHE(200)

Schema with Basic SPLASHE

genderMale	genderFemale	salaryMale	salaryFemale
ASHE(1)	ASHE(0)	ASHE(1000)	ASHE(0)
ASHE(0)	ASHE(1)	ASHE(0)	ASHE(2000)
ASHE(0)	ASHE(1)	ASHE(0)	ASHE(200)

Figure 3: SPLASHE instead of deterministic encryption.

by computing:

$$\mathsf{Enc}_k(m,i) := ((m - F_k(i) + F_k(i-1)) \bmod n, \{i\})$$

In other words, the ciphertext is a tuple (c, S), where c is an element of the group $\mathbb{Z}_n$ and S is a multiset of identifiers. Note that the ciphertext c consists of the plaintext value m plus some pseudo-random component, hence it appears to be random to anyone who does not know the secret key k.

To create the additive homomorphism, we define a special operation $\oplus$ for "adding" two ciphertexts:

$$(c_1, S_1) \oplus (c_2, S_2) := ((c_1 + c_2) \bmod n, S_1 \cup S_2)$$

That is, the group elements are added together and the multisets of identifiers are combined. To decrypt the ciphertext, Bob can simply compute

$$\mathsf{Dec}_k(c,S) := (c + \sum_{i \in S} (F_k(i) - F_k(i-1))) \bmod n$$

Thus, after the homomorphic operation,

$$\mathsf{Dec}_k(\mathsf{Enc}_k(m_1,i_1) \oplus \mathsf{Enc}_k(m_2,i_2)) = (m_1 + m_2) \bmod n$$

Figure 2 gives a high-level overview of ASHE in the context of Seabed. We show that the above scheme satisfies the standard notion of semantic (CPA) security in [36].

3.2 Optimizations for ASHE

The reader may wonder why the first element of the ciphertext is computed as $(m - F_k(i) + F_k(i-1)) \bmod n$ and not simply as $(m - F_k(i)) \bmod n$. The reason is that we have optimized ASHE for computing aggregations on large data sets. Suppose, for instance, that Alice wants to give Charlie a large table of encrypted values, with the intention that Charlie will later add up a range of these values and send them to Bob. Then Alice can simply choose the identifiers to be the row of numbers $(1, 2, \ldots, x)$. Later, if Bob receives an encrypted sum (c, S) with $S = \{i, \ldots, i+t\}$ (i.e., the sum of rows i to $i+t$), he can decrypt it simply by computing $(c + F_k(i+t) - F_k(i-1)) \bmod n$, since the other F_k values will cancel out. Thus, it is possible to decrypt the sum of a range of values by evaluating the PRF only twice, regardless of the size of the range.

Other optimizations including managing ciphertext growth and use of AES encryption support in hardware for efficient PRF computation are discussed in Section 4.

3.3 Basic SPLASHE

SPLASHE is motivated by frequency attacks on deterministic encryption [33]. Recall that, unlike ASHE, in deterministic encryption, there is only one possible ciphertext value for each plaintext value. This enables the server to perform equality checks but also reveals frequency of items. The attacker combines the frequency of ciphertexts with auxiliary information to decode them.

We begin by describing a basic version of our approach. Consider a column C_1 that can take one of d discrete values and let the value of C_1 in row t be $C_1[t]$. If we anticipate counting queries of the form `SELECT COUNT(`C_1`) WHERE` C_1`=x`, we can replace the column C_1 with a family of columns $C_{1,1}, \ldots, C_{1,d}$. When the value of $C_1[t]$ is v, we set $C_{1,v}[t] = 1$ and set $C_{1,w}[t] = 0$ for $w \neq v$. If the resulting columns are encrypted using ASHE, the ciphertexts will look random to the adversary, but it is nevertheless possible to compute the count: we can simply rewrite the above query to `SELECT SUM(`$C_{1,x}$`)` and then compute the answer using homomorphic addition.

A similar approach is possible for aggregations. Consider a pair of columns C_1 and C_2, where C_1 again takes one of d discrete values and C_2 contains numbers that we might later wish to sum up using a predicate on C_1 (and possibly other conditions). In other words, we anticipate queries of the form `SELECT SUM(`C_2`) WHERE` C_1`=x`. In this case, we can split C_2 into d columns $C_{2,1}, \ldots, C_{2,d}$. When $C_1[t] = v$, we set $C_{2,v}[t] := C_2[t]$ and set $C_{2,w}[t] := 0$ for $w \neq v$. C_1 and C_2 can then be omitted. Thus, the above query can be rewritten into `SELECT SUM(`$C_{2,x}$`)`, which can be answered using homomorphic addition. An example of SPLASHE is shown in Figure 3 for C_1 as `Gender` and C_2 as `Salary`.

3.4 Enhanced SPLASHE

Basic SPLASHE increases a column's storage consumption by a factor of d, which is expensive if d is large. Next, we describe an enhancement that addresses this.

Consider again a pair of columns C_1 (say, `country`) and C_2 (say, `salary`), where C_1 takes one of d discrete values and C_2 contains numbers that we might later wish to sum up using a predicate on C_1. Suppose k of the d values are common (e.g., a Canadian company with offices worldwide but with most employees located in `USA` or `Canada`; $k = 2$, $d = 196$). Then we can replace C_2 by $k + 1$ columns – one for each of the common values (`salaryUSA` and `salaryCanada`) and a single column for the uncommon values (`salaryOther`). Figure 4 shows an example. As before, for each row, we place the ASHE encrypted value of salary from C_2 in the appropriate salary column, while we fill the other k salary columns with ASHE-encrypted zeros. We then encrypt C_1 *deterministically* for each of the uncommon countries to enable equality checks against encrypted values.

At this point it is possible to compute aggregations on C_2 for all values v of C_1: if the value v is common (`USA` or `Canada`), we can compute a sum over the special column for v; otherwise we can select the rows where `country` in C_1 equals the deterministically encrypted value of v and compute the sum over `salaryOther`.

However, C_1 now is susceptible to frequency attacks. To prevent this, in C_1, we ensure that *all ciphertexts occur at the same frequency*. How is this possible? Note that the cells corresponding to common countries in C_1 were so far unused. We can reuse these cells to normalize the frequency count of the uncommon countries. For these reused cells, since the corresponding values in the `salaryOther` column are set to ASHE encrypted values of zero, this approach preserves correctness while preventing frequency attacks.

When is this approach possible? Let $n_1 \geq n_2 \ldots \geq n_d$ be the number of occurrences of each of the d values. Then the number of splayed columns should be chosen to be the minimum k such that $\sum_{i=1}^{k} n_i \geq \sum_{i=k+1}^{d} (n_{k+1} - n_i)$: this is because $\sum_{i=1}^{k} n_i$ are enough unused cells in column C_1 that can be used to make the number of occurrences of all non-splayed values at least n_{k+1}. Such a k will always exist; the more heavily skewed the distribution of values is, the smaller the k will be, and the more storage will be saved. This approach can be followed even if the exact number of occurrences is unknown; we do, however, need to know the distribution of the values.

Figure 4 shows an enhanced SPLASHE example with $k = 2$ and $d = 9$. Notice how the first six rows of the deterministically encrypted column have been reused to equalize the frequency of all elements in that column while still ensuring the correctness of aggregation queries on any of the country predicates.

Plaintext Schema

country	salary
USA	100000
USA	100000
Canada	200000
USA	300000
Canada	500000
Canada	800000
India	100000
India	100000
Chile	200000
Iraq	300000
China	500000
Japan	800000
Israel	130000
U.K.	210000

Schema with Enhanced SPLASHE

country	salaryUSA	salaryCanada	salaryOthers
DET(Chile)	ASHE(100000)	ASHE(0)	ASHE(0)
DET(Iraq)	ASHE(100000)	ASHE(0)	ASHE(0)
DET(China)	ASHE(0)	ASHE(200000)	ASHE(0)
DET(Japan)	ASHE(300000)	ASHE(0)	ASHE(0)
DET(Israel)	ASHE(0)	ASHE(500000)	ASHE(0)
DET(U.K.)	ASHE(0)	ASHE(800000)	ASHE(0)
DET(India)	ASHE(0)	ASHE(0)	ASHE(100000)
DET(India)	ASHE(0)	ASHE(0)	ASHE(100000)
DET(Chile)	ASHE(0)	ASHE(0)	ASHE(200000)
DET(Iraq)	ASHE(0)	ASHE(0)	ASHE(300000)
DET(China)	ASHE(0)	ASHE(0)	ASHE(500000)
DET(Japan)	ASHE(0)	ASHE(0)	ASHE(800000)
DET(Israel)	ASHE(0)	ASHE(0)	ASHE(130000)
DET(U.K)	ASHE(0)	ASHE(0)	ASHE(210000)

Figure 4: Enhanced SPLASHE example.

The reader can find a more detailed description of enhanced SPLASHE's security properties in our technical report [36]. Briefly, enhanced SPLASHE satisfies simulation-based security; the adversary learns only the number of rows in the database, and the number of infrequently and frequently occurring values.

3.5 Limitations

ASHE: Homomorphic encryption schemes have traditionally been defined with a *compactness* requirement, which says that the ciphertext should not grow with the number of operations that are performed on it. This is done to rule out trivial schemes: for instance, one could otherwise implement an additive "homomorphism" by simply concatenating the ciphertexts $Enc(m_1)$ and $Enc(m_2)$ and then have the client do the actual addition during decryption. ASHE does not strictly satisfy compactness, but the evaluator (the cloud) still does perform the bulk of the computation on ciphertexts; also, the techniques in Section 4 ensure that the length of ASHE's ciphertexts does not grow too much.

In terms of performance, growing ciphertexts can create memory stress at the workers. In the case of a system without encryption, the worker nodes only need enough memory to hold the dataset. When using ASHE, the workers need to have some extra memory to construct the ID lists. This should not be a big problem in practice: as we will show in Section 6, the overhead is small enough for real-world big data applications that involve billions of rows. Nevertheless, this extra memory requirement can become a problem if workers have very limited memory, or if the dataset is very large (e.g., if it has trillions of records).

SPLASHE: SPLASHE has three main drawbacks: (1) its requirement for a-priori knowledge of query workload

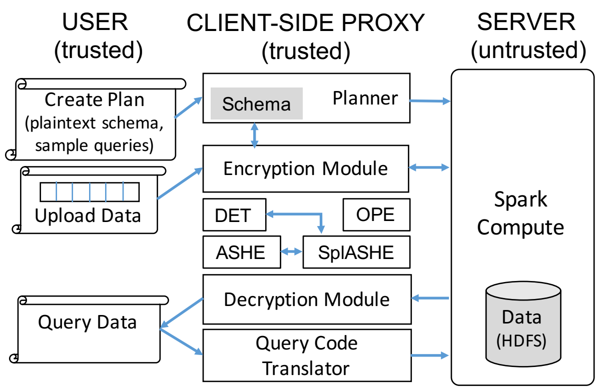


Figure 5: Seabed system design

or data distribution, (2) its difficulty in handling data with rapidly changing distribution, and (3) its storage overhead.

First, SPLASHE requires knowing what the expected query workload is. This is because we need to confirm that the splayed column will not participate in joins or inequality predicates – for such cases we need to fall back to deterministic encryption (DET). In addition, to get the storage reduction of enhanced SPLASHE, we need to know the distribution of values that a column can take. If this information is not available, only basic SPLASHE can be used.

Second, enhanced SPLASHE is most appropriate for columns whose distribution does not change dramatically. For columns whose distribution fluctuates significantly, data insertions will start skewing the distribution of the DET column (C_1 in our example) away from the uniform distribution SPLASHE constructs. This happens because a significant change in distribution will require reusing more cells than those available in the rows that were previously common. However, even in such an extreme case, SPLASHE is still better than using plain DET; DET reveals the exact distribution of values, whereas SPLASHE reveals a noised version of it.

Finally, both basic and enhanced SPLASHE increase storage needs. Section 6.6 shows that a real-world ad analytics database can be supported with enhanced SPLASHE at a storage overhead of about 10x.

4 Design

We now provide a functional overview of Seabed, and then describe each system component in more detail. For simplicity, we describe the design using the example of only one data source and one client. In practice, multiple data sources and users can share the same system as long as they share trust.

4.1 Roadmap

Figure 5 shows the major components of Seabed. A user interacts with the Seabed client proxy that runs in a trusted environment. The proxy in turn interacts with the untrusted Seabed server. As with previous systems, Seabed is designed to hide all cryptographic operations from users, so they interact with the system in the same way as they would with a standard Spark system. The user can issue three kinds of requests:

Create Plan: First, the user supplies a plaintext schema and a sample query set to the Seabed planner. The planner uses these and the procedure specified in Section 4.2 to determine the encryption schemes for the columns.

Upload Data: Next, the user sends plaintext data to the Seabed encryption module described in Section 4.3. The data is encrypted with the required encryption scheme and records are appended to the table stored in the Cloud. This is a continuing process; database insertions are handled in the same way.

Query Data: During analysis, the user sends a query script to the Seabed query translator, which modifies queries to run on encrypted data before sending them to the server (Section 4.5). The server runs the queries and responds to the proxy's decryption module (Section 4.6). After decryption and further processing (if any), the results are sent back to the user.

4.2 Data Planner

The data planner determines how to encrypt each column in the schema, given a list of sensitive columns by the user. The user also supplies a sample query set, which is used by the planner to decide on the encryption algorithms. In addition, to use enhanced SPLASHE, the user provides the number of distinct values each column can take and the frequency distribution of these values.

By parsing the sample query set, the planner first classifies each sensitive column as a *dimension*, a *measure*, or both. A measure is a column (e.g., `Salary`) over which a query computes aggregate functions, such as sum, average and variance. A dimension is a column (e.g., `Country`) that is used to filter rows based on a specified predicate before computing aggregates. After the classification, the planner uses the following strategies to determine which encryption schemes to use.

ASHE: If a sensitive measure is aggregated using linear functions, such as sum and average, we encrypt it using ASHE. If a sensitive measure is aggregated using quadratic functions (e.g., variance), we compute the square of the values on the client side and add it to the database as a separate column, so it can be used in computations on the server side. Whenever we use ASHE on a column, we give a unique ID to each row, which is used in the encryption as discussed in Section 3.1; to enable

Operation	Time (nanoseconds)
AES counter mode	47
Paillier encryption	5,100,000
ASHE encryption/decryption	12-24
Plain addition	1
Paillier addition	3800
Paillier decryption	3,400,000

Table 1: Cost of operations on a 2.2 GHz Xeon core.

compression, we assign consecutive row IDs. We choose a different secret key k for each new column we encrypt.

SPLASHE: If a sensitive dimension is used in filters, and if no query uses joins on this dimension, then the dimension is a candidate for SPLASHE. However, given the storage costs, we determine whether to use SPLASHE for the dimension as follows. First, we determine the measure columns that are used in conjunction with this dimension in the queries: only these measure columns need to be SPLASHE-encrypted. Based on this subset of measure columns, the planner uses the algorithm described in 3.4 to compute the storage overhead. Then, if a user specifies a maximum storage overhead, the planner prioritizes the dimensions that use SPLASHE based on their cardinality (lowest cardinal dimension first, in order to maximize protection against frequency attacks). We show how this approach works with a real dataset in Section 6.6.

DET or OPE: If a sensitive dimension cannot use SPLASHE – say, because it is used as part of a join – we warn the user and then use deterministic encryption (DET). If the dimension requires range queries in query filters, then we use order-preserving encryption (OPE). We require an OPE scheme that works on dynamic data and hence the OPE scheme of CryptDB [37] is not suitable in our case. We use the recent scheme from [18], which is efficient (based on any PRF) and has low leakage: for any two ciphertexts, in addition to the order of the two underlying plaintexts, it reveals the first bit where the two plaintexts differ and nothing more. For more details, please see our technical report [36].

Note that some queries (such as averages) cannot be directly executed on the server because they are not supported by Seabed's encryption schemes. In such cases, the Seabed planner borrows techniques from prior work [41] to divide the query into a part the server *can* compute (e.g., a sum and a count), and a part that the client/proxy will need to compute after decryption (e.g., the final division).

4.3 Encryption Module

The Encryption Module encrypts plaintext records into the encrypted schema. Note that ASHE encryption and decryption are quite lightweight compared to Paillier operations. As shown in Table 1, one AES counter operation (implemented using hardware support on a Intel Xeon 2.2GHz processor) takes 47 ns whereas one Paillier encryption takes 5.1 ms, a difference of *five orders of magnitude*. Hence, by using ASHE instead of Paillier, we reduce the encryption load on the client significantly.

We optimize ASHE encryption and decryption further by using a single AES operation to generate multiple ciphertexts. Each AES operation works on 128-bit vectors. Numeric data types are typically much smaller: 32-bit or 64-bit integers are common. One AES operation can therefore generate two or four pseudo-random numbers for 64-bit or 32-bit data types, respectively.

Also, note that unlike conventional cryptographic techniques, ASHE encryption and decryption are inherently parallelizable because multiple AES operations can be computed simultaneously in a multi-core environment. We therefore run a multi-threaded version of the encryption and decryption algorithm, and this further reduces latency.

If the system needs a way to revoke the access privileges of individual users, the proxy can additionally implement an access control mechanism, analogous to the approach in CryptDB. Typically, revocation is difficult when symmetric encryption schemes are used: once a symmetric key is shared, the only way to invalidate it is to re-encrypt the data. However, since the proxy handles all queries, it does not need to share the secret keys with the clients, so it can revoke or limit their access without re-encryption.

4.4 Query Translator

The goal of the Query Translator is to intercept the client's unmodified queries, and rewrite them in a way appropriate for the schema of the encrypted dataset. Our design follows the principles introduced by CryptDB and Monomi: we encrypt constants with the appropriate encryption scheme, and we replace operators with the custom functions that implement ASHE aggregation, or DET/OPE checks. One technical difference to the previous systems is that these operated on relational databases, so both the source and target language of the translator was SQL. However, Seabed works on Spark, so the target language is Scala and the Spark API.

The Seabed Query Translator makes three additions to the query rewriting process to accommodate the new encryption schemes it uses; we show examples for all three in Table 2. First, the schema of the encrypted dataset in Seabed includes an additional ID column. This column is necessary for ASHE aggregation, so the Query Translator preserves it even if the client has not explicitly done so in the projection fields of the original SQL query. That way, Seabed can support aggregation on the result of sub-queries. Second, for columns that use SPLASHE, Seabed follows the rules outlined in Section 3 to rewrite

Query type		Query
ID preservation	SQL	`SELECT sum(tmp.a) FROM (SELECT a FROM table WHERE b > 10) tmp`
	Spark API	`table.filter(x => x(2) > 10).map(x =>x(1)).reduce((x,y) => x+y)`
	Seabed	`table.filter(x => OPE.leq(x(2),Enc`$_{OPE}$`(10)).` `map(x =>(x(id), x(1))).reduce((x,y) => ASHE(x,y))`
SPLASHE	SQL	`SELECT count(*) FROM table WHERE a = 10`
	Spark API	`table.filter(x=>x(1) == 10).count()`
	Seabed	`table.map(x=>(x(id),x(3))).reduce((x,y)=>ASHE(x,y))`
Group-by optimization (and ID preservation)	SQL	`SELECT a, sum(b) FROM table GROUP BY a`
	Spark API	`table.map(x=>(x(1),x(2)).reduceByKey((x,y)=>x+y)`
	Seabed	`table.map(x=>(x(1)+":"+r.nextInt%10,(x(id),x(2))).` `reduceByKey((x,y)=>ASHE(x,y))`

Table 2: Examples of query translation. $x(1)$ corresponds to table column a, $x(2)$ to b, $x(3)$ to splayed a for value 10, and $x(id)$ to the identifier column used by ASHE.

Technique	Example	
	Integer/List	Encoding
Range encoding	[2...14,19...23]	[2-14,19-23]
Diff. encoding	[2,3,4,9,23]	[2,1,1,5,14]
Combination	[2...14,19...23]	[2-12,5-4]
VB-encoding	Encoded with minimum #bytes	

Table 3: ID list encoding techniques used in Seabed.

queries. This implies that the client has to maintain a small data structure with information about the splayed fields. Finally, if the client enables our group-by optimization, which is described in Section 4.5, the Query Translator may also modify the group-by fields of the query. This requires that the client maintains some state about the expected number of groups in a query result.

4.5 Seabed Server

Performing aggregations using ASHE requires the server to manage growing ciphertexts. This can result in need for large in-memory data structures and high bandwidth. We now describe how we optimize these overheads.

Reducing ID list size: To keep the size of the ID list small, we evaluated several integer list encoding techniques [31], including bitmaps [17], for good compression rates, low memory usage and high encoding speed. We eventually decided that a combination of the techniques listed in Table 3 were the most appropriate for Seabed. We begin with range encoding, which compresses contiguous sequences of integers by specifying the lower and upper bound. Next, we apply differential (Diff) encoding, which replaces the (potentially large) individual numbers with the (hopefully small) difference to the previous number; the result of this second step is labeled “Combination” in Table 3. Finally, we apply variable-byte (VB) encoding, which uses fewer bytes to represent smaller numbers.

Variable-byte (VB) and differential encoding (Diff) strike a nice balance between performance and compression and can be efficiently implemented in software. Range encoding, i.e. describing contiguous integers by specifying the bounds of their range, is not widely used in the literature because it can bloat up lists of non-contiguous integers. In Seabed, though, data is uploaded to the server with contiguous IDs, so range encoding can provide great benefits, especially for queries that select a large portion of a dataset. In Section 6.4, we show how combining VB, Diff, and range encoding reduces the size of the ID list and speeds up aggregation.

Reducing server-to-client traffic: Every Spark job consists of one driver node and several worker nodes. The workers send their partial results to the driver which then aggregates and sends the combined result to the client. To further reduce the size of ID lists, we applied standard compression. However, there are two options here: applying compression at the worker nodes or applying compression after aggregation at the driver node. The latter can lead to higher compression rates, but we found that this caused a bottleneck at the driver. Instead, we found that applying compression at each of the worker nodes benefits from parallelization and results in lower overall latency.

Handling group-by queries: Group-by queries are in general challenging for ASHE, because all row IDs are included in the final result, which can grow quite large. Moreover, using range encoding seems to incur unnecessary costs for group-by queries: when the result of a group-by query contains many groups, the ID lists of each group tend to be very sparse. As we noted earlier, range encoding is wasteful for sparse ID lists, so we decided to use only VB and Diff encoding for group-by queries.

Group-by queries lead to one more complication: when the number of groups in the result is small, the traffic between mapper and reducer workers becomes a bottleneck. There are two underlying reasons for this. First, with few groups, the ID list of each group becomes denser, and not using range encoding starts to show up. Second, when the number of groups is less than

Query set	Total	Purely on Server	Client Pre-processing	Client Post-processing	Two Round-trips
Ad Analytics	168,352	134,298	0	34,054	0
TPC-DS	99	69	2	25	3
MDX	38	17	12	4	5

Table 4: Different categories of queries that Seabed supports.

Dataset	Rows	Dimen-sions	Measu-res	Disk size (GB)			Memory size (GB)		
				NoEnc	Seabed	Paillier	NoEnc	Seabed	Paillier
Synthetic - Large	1.75B	-	1	35.4	70.4	521.1	84.7	121.9	638.6
Synthetic - Small	250M	-	1	5	9.8	74.2	12.1	17.7	91.4
BDB - Rankings	90M	1	2	7.9	12	58.3	18.6	28.1	80.4
BDB - User Visits	775M	8	2	194.9	287.5	673.6	581	832.5	1269.4
BDB - Query 4, Phase 2	194M	2	1	35	38.3	88.3	73.5	86.5	140
Ad Analytics	759M	33	18	132.3	142.45	176.3	1004	1027.3	1254.4

Table 5: Characteristics of our synthetic dataset, the Big Data Benchmark (BDB) and the Ad Analytics dataset (AdA).

the available workers, some reducers will remain idle in the reduce phase. This means that more data (because of denser ID lists) is shuffled between fewer workers (because of idle workers). This can create a bottleneck for very large datasets where ID lists are large.

To make use of more worker nodes in the reduce phase and to mitigate the above effect, we artificially increase the number of returned groups. We accomplish this by appending a random identifier to each value of the group-by column. For example (table 2), if a query returns 10 groups $\{g_1, \ldots, g_{10}\}$, and there are 100 workers available, then we can append a random identifier to the group-by column, which takes values from 0 to 9. This means that the result will contain $10 * 10 = 100$ groups $\{g_1{:}0, \ldots, g_1{:}9, \ldots, g_{10}{:}0, \ldots, g_{10}{:}9\}$, the computation will utilize all available workers in the reduce phase, and we will avoid the bandwidth bottleneck. Of course, the client has to perform the remaining aggregations to compute the sum of the actual groups (e.g., add results for groups $\{g_1{:}0, \ldots, g_1{:}9\}$ to get the result for group g_1). As a heuristic, we inflate the number of groups to the number of available workers when we expect fewer groups than workers.

4.6 Decryption Module

The Decryption Module uncompresses the ID lists, uses the techniques from Section 4.3 to calculate the pseudorandom numbers to add to the encrypted value, and returns the result to the user. If the query has some part that cannot be computed at the server, the Decryption Module can additionally perform that part before presenting the final answer to the user. Since we have assumed that the adversary is honest but curious, the Decryption Module performs no integrity checks; thus, an active adversary could return bogus data without being detected by Seabed itself.

The decryption cost of ASHE depends on the number of aggregated elements; this is different from Paillier, which requires only one decryption for each aggregate result. However, Paillier decryption is five orders of magnitude slower than ASHE decryption (Table 1), and the overall client decryption costs for Seabed remain smaller than Paillier (Section 6).

5 Applications

An important question is whether Seabed supports a wide range of big data analytics applications. To understand this, we performed three studies. First, we systematically analyzed two common interfaces that BI applications use at the back-end: MDX (the industry standard) and Spark. Second, we evaluated a month-long query log made on a custom-designed advertising analytics OLAP platform to determine how effectively Seabed can support the functionality of these systems. Finally, we analyzed the TPC-DS query set. Detailed results of our MDX/Spark analysis can be found in our technical report [36]. Briefly, the analysis revealed that Seabed's functionality support falls into four categories:

Support fully on the server: Seabed's encryption techniques can fully support operations with no client support. Examples of such operations are computing the sum, average, count, and min.

Support with client pre-processing: Seabed can support quadratic computation necessary for more complex analytics such as anomaly detection, linear regression in one dimension, and decision trees that are supported by Watson Analytics [8] and Tableau [7]. To support this, the Seabed client has to compute squared values of the necessary columns, and encrypt them with ASHE.

Support with client post-processing: All applications and APIs we studied allow users to specify arbitrary functions of data. When these functions are complex, Seabed cannot perform them at the server and data has to be post-processed at the client. This is similar to how Monomi splits queries into server- and client-side components.

Support with two client round-trips: Some queries require the client to compute an intermediate result, re-encrypt it and send it back to the server for further processing.

Table 4 shows the numbers of queries that fall into these categories for the three query sets we analyzed. We analyzed the MDX API/TPC-DS query set manually; for the ad analytics query set, we used heuristics based on the query structure. For Ad Analytics and TPC-DS, about 75-80% of the queries can be supported purely on the server. This implies that these query sets mostly use simple aggregation functions. About 20-25% need client-side support. The TPC-DS query set and MDX API have a few queries (5-15%) that require two round-trips.

6 Evaluation

In this section, we report results from our experimental evaluation of Seabed. Table 5 summarizes the datasets used in our experiments. We evaluate the system with microbenchmarks (Synthetic), an advertising analytics data workload and query set (AdA), and the AmpLab Big Data Benchmark (BDB).

Our evaluation has two high-level goals. First, we evaluate the *performance benefits* of Seabed over systems that use the Paillier cryptosystem. Second, we quantify the *performance and storage overhead* incurred by Seabed as compared to a system with no encryption.

6.1 Implementation and Setup

We built a prototype implementation of Seabed on the Apache/Spark platform [2] (version 1.6.0). We chose Spark because of its growing user-base and performant memory-centric approach to data processing. The server-side Seabed library was written in Scala using the Spark API. The Seabed client uses Scala combined with a C++ cryptography module for hardware accelerated AES (with Intel AES-NI instructions). We implemented Paillier in Scala using the `BigInt` class. Data tables are stored in HDFS using Google Protobuf [4] serialization. In total, our Seabed prototype consists of 3,298 lines of Scala and 2,730 lines of C++.

Our experiments were conducted on an Azure HDInsight Linux cluster. The cluster consists of tens of nodes, each equipped with a 16-core Intel Xeon E5 2.4 GHz processor and 112 GB of memory. Machines were running Ubuntu (14.04.4 LTS) and job scheduling was done through Yarn. In our experiments, we compare the following system setups:

NoEnc: Original Spark queries over unencrypted data,

Paillier: Modified Spark queries over encrypted data; measures are encrypted using Paillier, and dimensions with DET and/or OPE, and

Seabed: Modified Spark queries over encrypted data; measures are encrypted using ASHE, and dimensions with DET and/or OPE.

For our microbenchmarks, we generated a synthetic dataset (see Table 5). The NoEnc and Paillier datasets consist of one column of plaintext integers and 2048-bit ciphertexts, respectively. The ASHE dataset consists of two columns: an ID and an integer value encrypted with ASHE (IDs are contiguous). In order to model predicates that choose selected rows of a table, we use a parameter called *selectivity* that varies between 0 and 1 and use it to choose *each row* randomly with the corresponding probability. Note that this random selection model allows us to study the various system trade-offs in these schemes, e.g., the total length of ID lists, and it also enables us to understand the worst-case behavior. (At first glance, a query that selects all even or odd rows may appear to be the worst case for Seabed, since range encoding with such a non-contiguous set of IDs will double the size of the resulting ID list. However, in this case, the ID list is in fact highly compressible because the differences between consecutive IDs is always two, so stock compression techniques work very well.)

All experiments, unless otherwise mentioned, used 100 cores and 1.75 billion rows of input data. For end-to-end results, we place the client in one of the nodes in the same cluster as the server. Thus, by default, the client is connected by a high-speed, low-latency link to the server (TCP throughput of 2 Gbps). However, we also perform experiments by varying this bandwidth (using the `tc` command).

6.2 Microbenchmark: End to End Latency

We first compare end-to-end latency for the three approaches with varying input sizes (250 million to 1.75 billion rows). In Figure 6, we show the median latency after running 10 queries for each input size. For Seabed, we show two lines: one with selectivity 100% and the other with selectivity 50%. We shall show in Section 6.4 that the former gives best-case latency while the latter gives worst-case latency for Seabed. For NoEnc and Paillier, we use a selectivity of 100% (their performance is linear with respect to selectivity).

Figure 6(a) shows the results for NoEnc and Seabed.

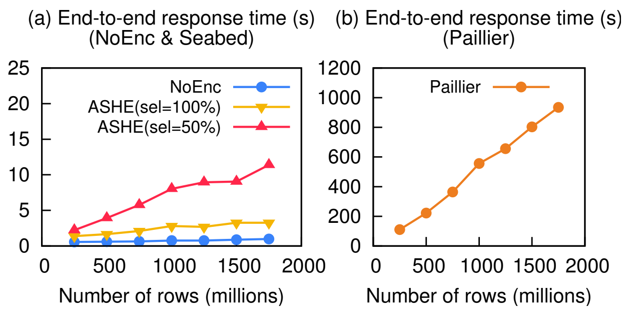


Figure 6: Median latency for aggregation vs data size.

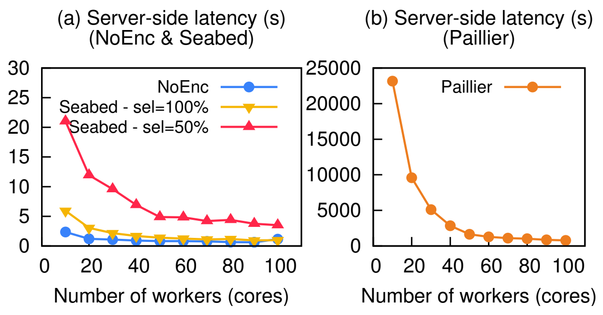


Figure 7: Median latency for aggregation vs cores.

NoEnc has a constant latency of approximately 0.6s. This is because addition is a simple operation and the overall latency is dominated by task creation costs. Seabed's aggregation is more complex, so latency for both Seabed selectivity 50% and 100% increases linearly with the dataset size. Nevertheless, the cost of aggregation in Seabed is still small even for large datasets, varying between 1.8s to 11s in the worst-case as the number of rows increase. On the contrary, Paillier results in a latency of over 1000s when aggregating 1.75 billion rows.

For Seabed selectivity 100%, about 80% of time is due to server-side compute, 20% is due to client-side decryption, and network latency is minimal. For Seabed selectivity 50%, the server-side contributes 55% of the latency, the decryption contributes 35% and network transfer contributes the remaining 10%.

We observed occasional stragglers, i.e., tasks that took longer to complete and delayed the entire job, for all three systems. The underlying cause of these stragglers was usually garbage collection being triggered at some node in the cluster. Paillier jobs took several hundreds of seconds to complete, so the comparative effect of stragglers was small. However, NoEnc and Seabed jobs took only few seconds at the server, so whenever there was a straggler task, the delay was more pronounced.

6.3 Microbenchmark: Server Scalability

One important aspect of big data systems is how they scale with larger clusters. Since using a larger cluster can only speed up the server side, we consider *server-side latency* as we evaluate Seabed's scalability. Fixing the dataset at 1.75 billion rows, we varied the number of cores from 10 to 100. Figure 7 shows how Seabed, NoEnc and Paillier scaled with the number of cores. NoEnc reached its best latency, which is approximately 1s, with 20 cores. Both Seabed selectivity 100% and Seabed selectivity 50% achieved their best latency of 1.35s and 8.0s respectively with only 50 cores. Even with 100 cores, Paillier's server latency was close to 1000s, which is more two orders of magnitude higher than Seabed's. This implies that, for large datasets, Paillier would require increasing the number of cores by orders of magnitude in order to achieve latencies that are comparable to Seabed. Seabed's overhead over NoEnc primarily comes from managing the ID lists. Next, we look into this in more detail.

6.4 Microbenchmark: Seabed Overhead

In this section we examine the server-side overheads incurred by Seabed's ASHE and the use of OPE.

ASHE list construction: For ASHE, the server manages ID lists using a variety of compression techniques (Section 4). In this experiment, we show how these compression techniques perform. The bitmap algorithms performed poorly, so we omit them here for brevity. We varied selectivity from 10% to 100%, and we measured the size of the ID list and the server-side response time of the query. We report the results in Figure 8(a) and (b).

Figure 8(a) suggests that range encoding is very effective in bounding the length of the ID list: without it, the size of ID list would keep increasing as the selectivity of a query increases, whereas with ranges the list size starts decreasing after selectivity 50%. After this, IDs start to become more dense and therefore more consecutive, leading to best-case compression at selectivity 100%. We can also see that the combination of VB and Diff-encoding is very effective in reducing the size of the ID list, and Deflate compression [6] further reduces the size of the list.

The performance hit incurred by each encoding method is depicted in Figure 8(b). To our advantage, we found that, in all cases except with Deflate optimized for high compression ratio, the better-performing algorithms also provided more compressed ID lists. Based on the above, we picked the following combination of encodings as the ID list construction algorithm in Seabed: Range-encoding, VB encoding, Diff-encoding, and Deflate compression (optimized for speed). This is what we used for all the other experiments.

OPE: The OPE scheme we use introduces some over-

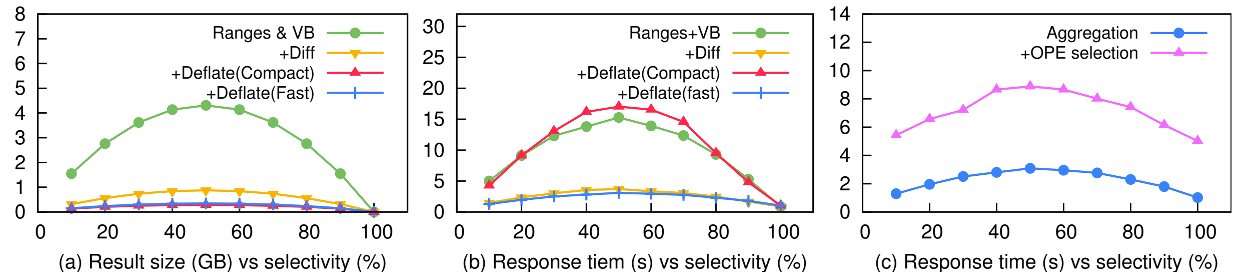


Figure 8: Result size and response time vs selectivity over 1.75 billion rows.

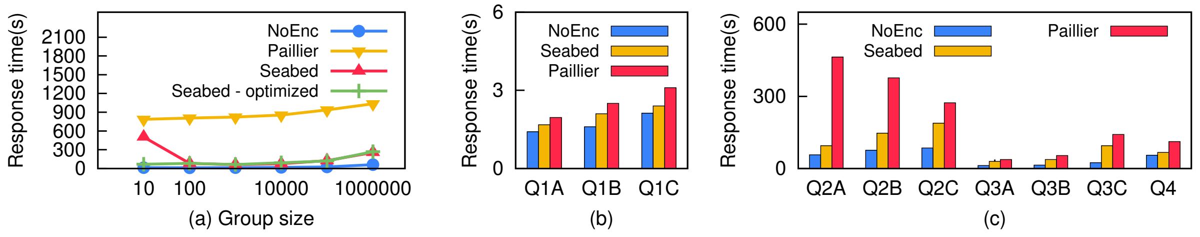


Figure 9: (a) Microbenchmark results for group-by queries. (b-c) Response time for the Big Data Benchmark queries.

head because comparison between OPE ciphertexts is not as fast as comparing two plaintext integers. This is because OPE comparison involves searching for the first bit position where two 64-bit integers differ.

To measure the cost of OPE, we used the same synthetic dataset as for ASHE with 1.75 billion rows, but we added one more integer column encrypted with OPE. We repeat the selectivity experiment above, but with the query performing an OPE comparison. Figure 8(c) indicates that OPE introduces more overhead, of about a factor of 5s, compared to the ASHE ID list construction.

6.5 Microbenchmark: Group-by

So far, we have evaluated only simple aggregation queries that involved minimal network communication: each Spark worker computes a sum and a compressed ID list per partition, and the reducers concatenate the lists into the final result. While aggregation is a major component of analytical query workloads, many queries also use the group-by operation, which causes more data to be shuffled across workers. In this section, we examine how Seabed performs for queries that involve group-by.

For this experiment, we used the synthetic dataset from the previous sections, but we added one more integer column. We then aggregated the value field while doing a group-by on the new column. We varied the number of groups from 10 to 1 million; Figure 9(a) shows the results.

The Seabed line shows the performance we get when we use VB and Diff-encoding for group-by queries. A very small number of groups in the result (10 in Fig. 9(a)) leads to increased latency because of the bandwidth bottleneck described in Section 4.5. The Seabed-optimized line shows that we can effectively deal with this inefficiency by artificially increasing the number of groups to 100 (Section 4.5).

Since all IDs are included in the result, Seabed group-by queries involve a significant amount of data shuffling. As a consequence, the benefits Seabed enjoys when compared to Paillier are lower. Yet, Seabed (optimized) does seem to be faster than Paillier by 5x to 10x. As the number of groups increases, Seabed's gain over Paillier drops from 10x to 5x. This is because the network shuffle time becomes a more significant part of the server response time. This indicates that Seabed will be less effective for group-by queries with a huge number of groups (hundreds of millions), something we observe in Section 6.6.

6.6 Ad-Analytics Workload

To assess the performance of Seabed on real-world data and queries, we evaluated it using the AmpLab Big Data Benchmark [1] and using a real-world large-scale advertising analytics application. We begin with a discussion of the latter.

For this series of experiments, we used data from an advertising analytics application deployed at an enterprise. This application is used by a team of experts for analytical tasks such as determining behavioral trends of advertisers, understanding ad revenue growth, and flagging anomalous trends in measures such as revenue and number of clicks. The data characteristics are shown in Table 5. We also obtained a set of queries that were per-

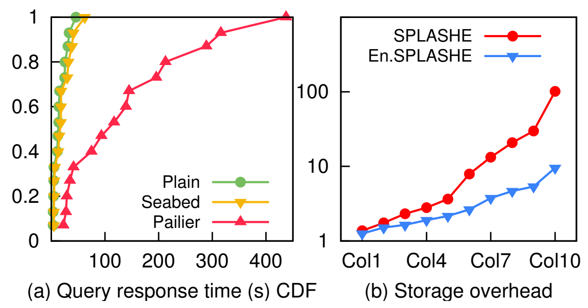


Figure 10: Seabed on the Ad Analytics workload: (a) query response-time CDF and (b) storage overhead due to SPLASHE.

formed for this application; this set consists of 168,352 queries issued between Feb 1, 2016 and Feb 25, 2016. The queries are all aggregations that calculate sums of various measures while grouping by timestamp (hour-of-day). The number of groups in a typical query is quite small, varying between 1 and 12 in most cases.

Performance: We first evaluated Seabed's performance on this dataset. We pick a set of 15 queries: five queries each for groups of size 1, 4, and 8. We ran each query ten times, and we calculated the median response time per query. All experiments were run with 100 cores.

Figure 10(a) shows the cumulative distribution function of response times for NoEnc, Seabed and Paillier. Seabed's response time ranges from 1.08 to 1.45 times that of NoEnc. The median response time for Seabed is 17.8s, whereas for NoEnc it is 13.8s. Thus Seabed's response time is only 27% higher than NoEnc's. On the other hand, the median response time for Paillier is 6.7× that of Seabed. To understand this result in more detail, we looked at the characteristics of the query responses. The average number of rows aggregated for a query across all groups was 210 million, the average size of the ID list was only 163.5KB, and the average number of AES operations required for decryption was roughly 26,000. This shows that there is a lot of contiguity of IDs in the ASHE ciphertext lists. Therefore, while queries could theoretically choose rows at random and thus create huge ID lists, our real-world dataset shows that this does not necessarily happen in practice: the data is stored in a certain order, and Seabed benefits from that order.

In all our experiments, the Seabed client used a high-bandwidth link to connect to the server. To measure the effect of lower-bandwidth and higher-latency links, we artificially changed the network bandwidth/latency between server and client to 100Mbps/10ms and 10Mbps/100ms. This increased the median response time by only 1% in the former case and 12% in the latter case, as the ID lists that need to be transferred are quite small.

Storage: We also used this dataset to quantify SPLASHE's overall storage overhead. Through conversations with operators, we determined that 10 out of 33 dimensions and 10 out of 18 measures require encryption. We used the procedure outlined in Section 3.4 to calculate the storage overhead for these 10 dimensions.

Figure 10(b) shows the cumulative storage overhead for each of the 10 dimensions in our dataset, sorted by the number of unique values in the dimension. The graph shows that if we restrict the storage overhead to a factor of two, we can encrypt only one dimension with Basic SPLASHE, whereas we can encrypt two dimensions with Enhanced SPLASHE. With a storage overhead of three, we can encrypt only three dimensions with Basic SPLASHE, whereas we can encrypt 6 with Enhanced SPLASHE. In this case, roughly 92% of all queries involve at least one column that uses enhanced SPLASHE.

6.7 AmpLab Big Data Benchmark

The AmpLab benchmark includes four types of queries (scan, aggregation, join and external script). Some of them come in different variants based on the result/join size, so there are ten queries in total. For this experiment we used 32 cores and loaded the entire Big Data Benchmark dataset (table 5) into the workers' memory. We measured the time to perform the query and store the results back into cache memory. Since the Big Data Benchmark is not designed for interactive queries, most of the result sets are huge and cannot fit into one machine's memory. Hence, for this section we do not measure the client-side cost of any of the compared systems.

We had to make a few simplifications to the query set in order to support it. Queries 2 and 4 require substring-search over a column and a text file, respectively. Existing searchable encryption techniques do not efficiently support this operation. Hence we simplified query 2 by matching over deterministically encrypted prefixes, and we simplified query 4 by keeping the text file as plaintext. Query 3 involves sorting based on aggregated values; since this can only be done on the client, and given that we measured only server-side overhead in this experiment, we omitted the sorting step.

Figure 9(b) shows the results. Query 1 does not use group-by or aggregation, so all tested systems had much faster response times. Both Seabed and Paillier were slower than NoEnc because of OPE overheads. On the remaining queries Seabed was consistently faster than Paillier, though not as much as we had shown in Sections 6.2 and with the Ad Analytics workload. This is because the queries results contained millions of groups and, as we saw in Section 6.5, Seabed is slower on result sets with a very small or a very large number of groups. Nevertheless, the results show that Seabed is better than

Paillier even for these workloads and is close to NoEnc performance for most queries.

7 Related Work

Homomorphic Encryption. Homomorphic encryption allows computations to be performed on encrypted data such that the computed result, when decrypted, matches the result of the equivalent computation performed on unencrypted data. The first construction of a fully homomorphic scheme that allows arbitrary computations on encrypted data was shown in [23]. However, fully homomorphic schemes are far from practical even today. For example, the amortized cost of performing AES encryption homomorphically is about 2s [25] but this is still 10^8 times slower than AES over plain text (Section 4).

There are also partially homomorphic schemes that allow selected computations on encrypted data. For example, Paillier [35] allows addition of encrypted data while BGN [16] supports one multiplication and several additions. However, these schemes incur significant cost in terms of both computation and storage space. Algorithms to reduce storage overhead by packing multiple integer values into a single Paillier encrypted value are proposed in [22] and implemented in [41].

Encrypted databases. CryptDB [37] leverages partially homomorphic encryption schemes to support SQL queries efficiently over encrypted data, and Monomi [37] introduced a split client-server computation model to extend support for most of the TPC-H queries over encrypted data. However, as we show in this paper, the partially homomorphic encryption schemes used in CryptDB and Monomi are not efficient enough to support interactive queries when applied to large datasets.

Trusted hardware. Hardware support for trusted computing primitives, such as Intel SGX [32], secure coprocessors [27], and FPGA-based solutions [9], are available today. These solutions allow client software to execute in the cloud without providing visibility of client data to the cloud OS. Several prior systems – such as Cipherbase [9], TrustedDB [11], M2R [21] and VC3 [38] – rely on secure trusted hardware to provide privacy-preserving database or MapReduce operations in the cloud.

The use of trusted hardware has the potential to provide secure computations at minimal performance overhead. However the client has to trust that the hardware is free of errors, bugs, or backdoors. It is difficult to confirm that this is indeed the case, since errors can be introduced in both the design of the hardware and in the fabrication process, which is frequently outsourced [28]. In fact, hardware backdoors have been found in real-world military-grade hardware chips [39], and hardware trojan detection is an active research field in the hardware community [14]. We believe that it is useful to develop alternatives that rely only on cryptographic primitives.

Frequency attacks on property-preserving encryption. Property-preserving encryption schemes by definition leak a particular property of the encrypted data. For example, deterministic encryption [12] leaks whether two ciphertexts are equal, and order-preserving encryption [15] leaks the order between the ciphertexts. Naveed et al. [33] used auxiliary information and frequency analysis to show that one can infer the plain text from ciphertexts that have been encrypted using such property-preserving encryption schemes.

8 Conclusion

We have described Seabed, a system for performing Big Data Analytics over Encrypted Data. We have introduced two novel encryption schemes: ASHE for fast aggregations over encrypted data, and SPLASHE to protect against frequency attacks. Our evaluation on real-world datasets shows that ASHE is about an order of magnitude faster than existing techniques, and that its overhead compared to a plaintext system is within 45%.

9 Acknowledgements

We would like to thank Sriram Rajamani, Kapil Vaswani and Chandu Thekkath for helpful suggestions on early versions of this manuscript. We would also like to thank the OSDI reviewers for their constructive feedback and Bryan Ford for shepherding this paper. This work was supported in part by NSF grants CNS-1513694 and CNS-1054229.

References

[1] AmpLab Big Data Benchmark. `https://amplab.cs.berkeley.edu/benchmark/`.

[2] Apache Spark. `http://spark.apache.org/`.

[3] Big data benchmark. `https://amplab.cs.berkeley.edu/benchmark/`.

[4] Google Protocol Buffers. `https://developers.google.com/protocol-buffers/`.

[5] PowerBI. `https://powerbi.microsoft.com/en-us/features/`.

[6] RFC - DEFLATE Compressed Data Format Specification. `https://tools.ietf.org/html/rfc1951`.

[7] Tableau Online. http://www.tableau.com/products/cloud-bi.

[8] Watson Analytics. http://www.ibm.com/analytics/watson-analytics/us-en/.

[9] A. Arasu, S. Blanas, K. Eguro, R. Kaushik, D. Kossmann, R. Ramamurthy, and R. Venkatesan. Orthogonal security with cipherbase. In *Proc. of CIDR*, 2013.

[10] ARM. ARM Security Technology Building a Secure System using TrustZone Technology. ARM Technical White Paper, 2009.

[11] S. Bajaj and R. Sion. Trusteddb: A trusted hardware-based database with privacy and data confidentiality. *IEEE Transactions on Knowledge and Data Engineering*, 26(3):752–765, 2014.

[12] M. Bellare, A. Boldyreva, and A. ONeill. Deterministic and efficiently searchable encryption. In *Proc. CRYPTO, 2007*.

[13] R. Bhagwan, R. Kumar, R. Ramjee, G. Varghese, S. Mohapatra, H. Manoharan, and P. Shah. Adtributor: Revenue debugging in advertising systems. In *Proc. USENIX NSDI*, 2014.

[14] S. Bhasin and F. Regazzoni. A survey on hardware trojan detection techniques. In *Proc. IEEE ISCAS*, 2015.

[15] A. Boldyreva, N. Chenette, Y. Lee, and A. Oneill. Order-preserving symmetric encryption. In *Proc. EUROCRYPT, 2009*.

[16] D. Boneh, E.-J. Goh, and K. Nissim. Evaluating 2-dnf formulas on ciphertexts. In *Theory of cryptography*, pages 325–341. Springer, 2005.

[17] S. Chambi, D. Lemire, O. Kaser, and R. Godin. Better bitmap performance with roaring bitmaps. *Software: Practice and Experience*, 2015.

[18] N. Chenette, K. Lewi, S. A. Weis, and D. J. Wu. Practical order-revealing encryption with limited leakage. In *Proc. FSE, 2016*.

[19] V. Costan and S. Devadas. Intel sgx explained. Technical report, Cryptology ePrint Archive, Report 2016/086, 2016. http://eprint.iacr.org.

[20] S. Davenport and R. Ford. Sgx: the good, the bad and the downright ugly. *Virus Bulletin*, 2014.

[21] T. T. A. Dinh, P. Saxena, E.-C. Chang, B. C. Ooi, and C. Zhang. M2r: Enabling stronger privacy in mapreduce computation. In *Proc. USENIX Security, 2015*.

[22] T. Ge and S. Zdonik. Answering aggregation queries in a secure system model. In *Proc. VLDB, 2007*.

[23] C. Gentry. Fully homomorphic encryption using ideal lattices. In *Proc. ACM STOC*, 2009.

[24] C. Gentry, S. Halevi, and N. P. Smart. Homomorphic evaluation of the aes circuit. In *Proc. CRYPTO, 2012*.

[25] C. Gentry, S. Halevi, and N. P. Smart. Homomorphic evaluation of the AES Circuit (Updated Implementation), 2015. https://eprint.iacr.org/2012/099.pdf.

[26] O. Goldreich and R. Ostrovsky. Software protection and simulation on oblivious rams. *Journal of the ACM*, 43(3):431–473, 1996.

[27] IBM Corporation. IBM Systems cryptographic hardware products. http://www-03.ibm.com/security/cryptocards/.

[28] F. Imeson, A. Emtenan, S. Garg, and M. Tripunitara. Securing computer hardware using 3d integrated circuit (ic) technology and split manufacturing for obfuscation. In *Proc. USENIX Security*, 2013.

[29] K. Kambatla, G. Kollias, V. Kumar, and A. Grama. Trends in big data analytics. *Journal of Parallel and Distributed Computing*, 74(7):2561–2573, 2014.

[30] G. Lee, J. Lin, C. Liu, A. Lorek, and D. Ryaboy. The unified logging infrastructure for data analytics at twitter. In *Proc. VLDB, 2012*.

[31] D. Lemire and L. Boytsov. Decoding billions of integers per second through vectorization. *Software: Practice and Experience*, 45(1):1–29, 2015.

[32] F. McKeen, I. Alexandrovich, A. Berenzon, C. V. Rozas, H. Shafi, V. Shanbhogue, and U. R. Savagaonkar. Innovative instructions and software model for isolated execution. In *HASP@ ISCA*, page 10, 2013.

[33] M. Naveed, S. Kamara, and C. V. Wright. Inference attacks on property-preserving encrypted databases. In *Proc. ACM CCS*, 2015.

[34] R. Ostrovsky. Efficient computation on oblivious rams. In *Proc. ACM STOC*, 1990.

[35] P. Paillier. Public-key cryptosystems based on composite degree residuosity classes. In *Proc. EUROCRYPT, 1999.*

[36] A. Papadimitriou, R. Bhagwan, N. Chandran, R. Ramjee, A. Haeberlen, H. Singh, A. Modi, and S. Badrinarayanan. Big Data Analytics over Encrypted Datasets with Seabed, Technical Report MS-CIS-16-08, University of Pennsylvania, 2016.

[37] R. A. Popa, C. Redfield, N. Zeldovich, and H. Balakrishnan. CryptDB: protecting confidentiality with encrypted query processing. In *Proc. ACM SOSP*, 2011.

[38] F. Schuster, M. Costa, C. Fournet, C. Gkantsidis, M. Peinado, G. Mainar-Ruiz, and M. Russinovich. VC3: Trustworthy data analytics in the cloud using SGX. In *Proc. IEEE Security and Privacy, 2015.*

[39] S. Skorobogatov and C. Woods. Breakthrough silicon scanning discovers backdoor in military chip. In *Proc. International Workshop on Cryptographic Hardware and Embedded Systems*. Springer, 2012.

[40] R. Sumbaly, J. Kreps, and S. Shah. The big data ecosystem at LinkedIn. In *Proc. ACM ICMD*, 2013.

[41] S. Tu, M. F. Kaashoek, S. Madden, and N. Zeldovich. Processing analytical queries over encrypted data. In *Proc. VLDB*, 2013.

Non-intrusive Performance Profiling for Entire Software Stacks based on the Flow Reconstruction Principle

Xu Zhao, Kirk Rodrigues, Yu Luo, Ding Yuan, Michael Stumm
University of Toronto

Abstract

Understanding the performance behavior of distributed server stacks at scale is non-trivial. The servicing of just a single request can trigger numerous sub-requests across heterogeneous software components; and many similar requests are serviced concurrently and in parallel. When a user experiences poor performance, it is extremely difficult to identify the root cause, as well as the software components and machines that are the culprits.

This paper describes Stitch, a non-intrusive tool capable of profiling the performance of an entire distributed software stack solely using the unstructured logs output by heterogeneous software components. Stitch is substantially different from all prior related tools in that it is capable of constructing a system model of an entire software stack without building any domain knowledge into Stitch. Instead, it automatically reconstructs the extensive domain knowledge of the programmers who wrote the code; it does this by relying on the Flow Reconstruction Principle which states that programmers log events such that one can reliably reconstruct the execution flow a posteriori.

1 Introduction

Understanding the performance behavior of distributed server stacks at scale is non-trivial. Many incoming requests are serviced in parallel, and each such request may trigger multiple sub-requests on various software components spread out over many hosts. For example, a simple Hive query may involve a YARN Resource Manager, numerous YARN Application and Node Managers, several MapReduce tasks, and multiple HDFS servers.

Numerous tools have been developed to help identify performance anomalies and their root causes in these types of distributed systems. The tools have employed a variety of methods, all of which have significant limitations. Many methods require the target systems to be instrumented with dedicated code to collect information [1, 2, 9, 11, 24, 26, 27, 34]; as such, they are intrusive and often cannot be applied to legacy or third-party components. Other methods are non-intrusive and instead analyze already existing system logs; they either use machine learning approaches to identify anomalies [25, 38] or they rely on static code analysis [42]. Approaches that use machine learning techniques cannot understand the underlying system behavior and thus may not help identify the root cause of each anomaly. Approaches that require static code analysis are limited to components where such static analysis is even possible, and they are unable to understand the interactions between different software components.

We present a new approach for obtaining and presenting information useful for identifying performance issues and their causes in large distributed server stacks. Our method focuses on objects, their interactions, and their hierarchical relationships. Using pattern matching techniques on existing logs alone, we are able to identify which objects are participants in each logged event, which objects are of the same type, and how objects of different types relate to one another, allowing one to infer execution structure. Our method is non-intrusive and does not require static code analysis, yet enables complex cross-component performance analysis regardless of the programming languages used.

We have created a tool that extracts information from standard logs and is capable of visually displaying individual objects over their lifetimes showing how and when objects interact with each other. The gathered information on object relationships allows our tool to initially display high-level object instances (e.g., a Hive query) that can then be drilled down to view lower-level instances (e.g., HDFS blocks, MapReduce tasks, or containers). This hierarchical approach to displaying information is critical given the overwhelming number of events captured in logs and the number of objects involved. With a set of realistic sample scenarios, we show in Section 5 that such a tool can be useful for identifying performance anomalies and their root causes. Section 2 provides a motivating example.

This paper makes the following specific contributions:

- Our approach is the first to be able to construct a system model of an entire software stack without any built-in domain knowledge.
- We propose the first non-intrusive method that enables diagnosis of complex cross-component failures.
- Our method focuses on objects, their relationships and interactions as a way to deal with complexity (as opposed to focusing on events). Complexity is managed, for example, by initially displaying only high-level objects until a user decides to drill down on target objects.

Our approach has two limitations. Firstly, it is not able to directly identify causal relationships between events. However, we show that having this information is not necessary to debug complicated performance issues. Secondly, the efficacy of our approach relies on the quality of logging in the analyzed system. Specifically, our approach relies on the **Flow Reconstruction Principle**:

> programmers will output sufficient information to logs so as to be able to reconstruct runtime execution flows after the fact;

and more specifically:

> programmers will insert, for each important event, a log printing statement that outputs the identifiers[1] of all relevant objects involved with the event in order to be able to reconstruct execution flows a posteriori.

Inserting such log statements is a widely followed practice. As we show in Section 5, Hive/Hadoop, Spark, OpenStack, and even syslog [37] logs exhibit these properties. Many object identifiers, like process ID and thread ID, are automatically output by the underlying logging libraries for each event [21, 30, 35, 37].

The Flow Reconstruction Principle applies because a post mortem analysis is typically performed after each failure and the programmer will be asked to reconstruct what exactly transpired up to the point of failure in order to identify the root cause. This is non-trivial given the amount of concurrency and parallelism prevalent in scalable distributed server stacks. Hence, programmers will insert log statements to allow them to reconstruct how the failure occurred. Specifically they will:

- log a sufficient number of events — even at default logging verbosity — at critical points in the control path so as to enable a post mortem understanding of the control flow leading up to the failure.
- identify the objects involved in the event to help differentiate between log statements of concurrent/parallel homogeneous control flows. Note that this would not be possible when solely using constant strings. For example, if two concurrent processes, when opening a file, both output "opening file", without additional identifiers (e.g., process identifier) then one would not be able to attribute this type of event to either process.
- include a sufficient number of object identifiers in the same log statement to unambiguously identify the objects involved. Note that many identifiers are naturally ambiguous and need to be put into context in order to uniquely identify an object. For example a thread identifier (tid) needs to be interpreted in the context of a specific process, and a process identifier (pid) needs to be interpreted in the context of a specific host; hence the programmer will not typically output a tid alone, but always together with a pid and a hostname. If the identifiers are printed separately in multiple log statements (e.g., hostname and pid in one log statement and tid in a subsequent one) then a programmer can no longer reliably determine the context of each tid because a multi-threaded system can interleave multiple instances of these log entries.[2]

Programmers have a key advantage when using log messages to reconstruct the paths taken by the system when servicing requests: they can interpret the meaning of constant strings, and many identifiers contain string sequences that have meaning to the programmer. An automated tool does not have this advantage. As such, we disregard constant strings and do not attempt to extract semantics from object identifiers. Instead we extract information about objects by analyzing various patterns that exist in the logs. Our approach consists of the following five steps: (1) extract from the log messages the object identifiers; (2) associate each extracted object identifier with an object type; (3) identify how each object type is related to the other object types with respect to participating in the same event; (4) identify the specific object instances that are involved in each event; and (5) identify execution structure and hierarchy between objects. Using the extracted information, we are able to display the hierarchy of objects in play when servicing requests. We are also able to visually display objects along a timeline, as shown in the next section.

With the extracted hierarchy information, the visual display initially shows only the highest-level objects,

[1] In this paper, 'identifier' refers to the variable value that can be used to differentiate objects. Examples of identifiers include thread ID, process ID, file names, and host names. Examples of non-identifiers include the value of a counter or CPU usage statistics. Note that the counter itself is an object, but its value is not an identifier because it is not intended to be used to differentiate different counter instances.

[2] Note that the validity of this principle hinges on each log statement being thread safe. All of the widely used logging libraries we examined, including syslog [37], log4j [21], java.util.logging [16], SLF4J [35], log4cpp [20], Boost.log [5], and Python's logging module [30] are thread-safe.

but the user can selectively drill down incrementally by clicking on any of the objects of interest to expose more details at the lower-level. This enables identification of performance bottlenecks and analysis of potential root causes. A controlled user study showed that with Stitch, developers were able to speed up the analysis and debugging process by a factor of 4.6 compared to when they were restricted to using raw logs only. Our evaluation of how well Stitch is able to reconstruct a system model from logs showed that Stitch could do so with 96% accuracy when applied to Hive stack, Spark stack, and OpenStack logs produced by 200 nodes as well as logs from our production server stacks.

2 Motivating Example

We demonstrate the usefulness of Stitch using a real-world, user reported performance anomaly [31] that manifests itself across multiple software layers. Some Hive users reported that they occasionally experienced much longer than normal job completion times. We reproduced the anomaly on a three node cluster. We evaluated Stitch's effectiveness in a controlled user study comprising 14 experienced programmers, where half of the users were asked to debug with Stitch while the other half only had access to the raw logs. They were given 45 minutes. (Section 5.1 provides more details of the user study and other cases we used.)

On average, the users who did not use Stitch spent 38 minutes debugging this case whereas Stitch users only spent 12 minutes – a speedup factor of 3. In fact, six of the seven users not using Stitch failed to diagnose the anomaly whereas all seven Stitch users were successful. We first describe the users' experiences debugging without Stitch before discussing how Stitch helps.

Debugging without Stitch. Users in this control group immediately started grep'ping, awk'ing, and perl'ing the log files. Primarily, they took one of two approaches:

- "Bottom-up," where they searched for keywords such as "Error", "Warn", or "slow", or
- "Top-down," where they tried to understand the high-level hierarchy of the system before focusing on a particular lead.

The users who took a bottom-up approach almost entirely ended up in a wild-goose chase. For example, searching for "Error" and "Warn" returned log messages about non-zero container exit codes and failed container deallocations, neither of which pertained to the root cause. Three users stayed on this path with tunnel-vision, eventually reaching the time limit (45 minutes) without having gotten anywhere close to the root cause.

The three users who took a top-down approach got closer to finding the root cause, but none of them were able to identify it within the time limit. These users started by examining the latency of each Hive query in order to identify the slowest one. However, drilling down further proved to be daunting: users needed to follow the control path from Hive to the MapReduce layer to identify the map and reduce tasks created to process the query, then they had to carefully compare the timestamps of each task's first and last log message to identify the task-level bottleneck.

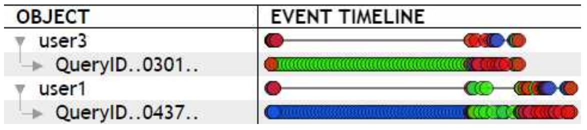


Figure 1: Stitch's timeline graph. Each line represents an object with its IDs listed in the left panel. Users can drill down to objects at the next level by clicking on the object. Each circle in the right panel represents an individual event, and its color indicates the node where the corresponding log message was output. Only two out of many queries are shown.

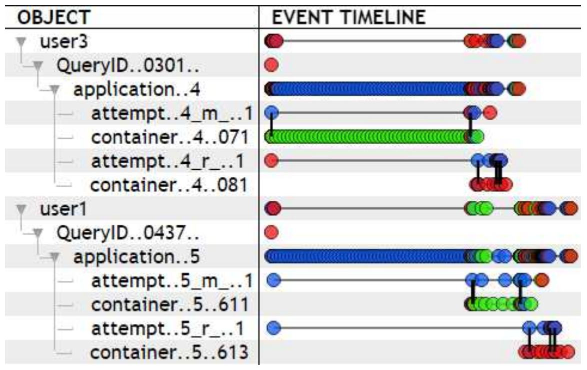


Figure 2: Stitch's timeline graph expanded to show more levels of the hierarchy. Clearly visible is that `user1`'s jobs start processing as soon as `user3` releases its containers. The vertical lines show the interactions among objects, which are inferred from the events that included multiple objects.

Even if the user could identify the slowest map/reduce task, the task's numerous interactions with other components makes it even more difficult to determine why it is slow. For example, a grep on particular (slow) map task may return over 500 log messages concerning its dealings with YARN containers, HDFS blocks, etc. Stitch serves to automate this process, freeing the user to focus on *debugging* rather than reconstructing the request.

Debugging with Stitch. Figure 1 shows Stitch's web-based GUI for this case. Objects are organized hierarchically allowing users to understand the system's structure as they drill down on each object. The event timeline shows the events where the object ID appeared such that the first and last event can be used to infer the object's lifetime. Thus, a user can immediately determine that Query "0437" has the longest execution time and drill down to investigate.

Figure 2 shows the interface following the expansion of both queries down to the level of map/reduce tasks. Under each query is the YARN application[3] created to process it, and under each application are the map/reduce task attempts and containers spawned to process the request. It shows that user1's Query 0437 has attempts that were created early on but only received containers much later. This allocation seems to correspond with the release of a container in user3's Query 0301, suggesting allocation was being serialized across users.

This serialization was caused by a bug in YARN's Capacity Scheduler which incorrectly limited all users' resource usage whenever a single user reached his or her limit. The study participants were not required to find this bug in the source since the study was limited to information that could be garnered from the logs alone. Instead they were only asked to point out that the serialized container allocation being the cause, which is conceivably the key step for the developers to diagnose the bug.

3 System Stack Structure Graph

In this section, we describe the algorithm Stitch uses to extract information from the logs necessary to identify objects, their interactions and their hierarchical relationships. The algorithm outputs a System Stack Structure (S^3) Graph – a directed acyclic graph (DAG), where each node represents an object type and each directed edge captures a hierarchical relationship between a high-level object type (parent) and a low-level object type (child).

We treat each logged *event e* as a set of *identifiers*, $id_{e1}..id_{en}$. We extract object identifiers by disregarding static substrings and applying a number of heuristics; e.g., we disregard variables preceded by common non-identifier types (e.g., "size") or succeeded by units (e.g., "ms"). How we extract IDs may seem somewhat ad hoc, but it works; ID parsing is considered a solved problem in the industry using tools such as Splunk [36], VMWare LogInsight [22] or Logstash [23]. Examples of IDs we extract this way include machine IP addresses, process IDs, thread IDs, and filenames. Note that the extracted IDs are often ambiguous until they obtain a context within which they can be interpreted; e.g., a process ID is unambiguous only if interpreted within the context of a specific host.

Each identifier is of a *type*, which is the type of the object it represents (e.g., a process, a thread, an IP address, a host name). In this section we represent identifiers in lowercase and their types in uppercase; e.g., both $host_1$ and $host_2$ are of type HOST. Stitch identifies the type by applying a number of heuristics that identify, for example, common static strings surrounding identifiers, common static substrings within identifiers, or the structure of the identifiers. Note that Stitch does not attempt to understand or identify what the actual type is (i.e., IP address, pid, filename, etc.), but simply differentiates between types abstractly (i.e., $TYPE_A$, $TYPE_B$, $TYPE_C$, etc.).

We say that two objects, obj_i and obj_j, are *correlated*, represented as $\text{obj}_i \sim \text{obj}_j$, if both objects were participants in the same logged event, meaning both of their IDs appeared in the same log message. We further define that obj_i *subsumes* obj_j, or $\text{obj}_i \vdash \text{obj}_j$, if and only if: (1) $\text{obj}_i \sim \text{obj}_j$, and (2) obj_j is not correlated with any other object of the same type as obj_i. For example, in Hive, user u_i subsumes query q_k because u_i will submit many different queries (including q_k), yet two queries with the same name will not typically be submitted by different users since each query is assigned a globally unique ID based on its timestamp and global order.

For all of the object types we identified, $T_{1..t}$, we categorize the relationship between each possible pair $(T_I, T_{J \neq I})$ as one of (*i*) empty, (*ii*) 1:1, (*iii*) 1:n, or (*iv*) m:n. We use this categorization to, in subsequent steps, help identify objects unambiguously and to identify the system stack object structure. The relationship is empty if object IDs of the two types never appear in the same log message. The relationship is 1:1, i.e., $T_I \equiv T_J$, if it is not empty, and $\forall$ $\text{obj}_i \in T_I, \forall \text{obj}_j \in T_J, \text{obj}_i \sim \text{obj}_j \Rightarrow (\text{obj}_i \vdash \text{obj}_j) \wedge (\text{obj}_j \vdash \text{obj}_i)$; for example, IP_ADDR $\equiv$ HOST if there is no IP remapping. It is 1:n, i.e., $T_I \rightarrow T_J$, if it is not empty or 1:1, and $\forall \text{obj}_i \in T_I, \forall \text{obj}_j \in T_J, \text{obj}_i \sim \text{obj}_j \Rightarrow \text{obj}_i \vdash \text{obj}_j$. Finally, the relationship is m:n, i.e., $T_I \bowtie T_J$, if and only if $\exists \text{obj}_i \in T_I, \exists \text{obj}_j \in T_J$, s.t. $\text{obj}_i \sim \text{obj}_j$ while $\text{obj}_i \nvdash \text{obj}_j$ and $\text{obj}_j \nvdash \text{obj}_i$.

The size of the logs being used for the analysis needs to be sufficiently large for us to be able categorize relationships correctly. If the size is too small, then some of the type relationships might be miscategorized; e.g., (USER, QUERY) will be categorized as 1:1 instead of 1:n if the log spans the processing of only one query. In contrast, logs spanning too large a time frame may also cause miscategorizations, at least theoretically; e.g., (USER, QUERY) might be categorized as m:n if the query ID wraps around. However, mature distributed systems like Hadoop, Spark, and OpenStack use universally unique identifier (UUID) [18] libraries to assign key identifiers. Therefore, the likelihood of identifier reuse is extremely low [18].

We can intuitively draw certain conclusions about a pair of identifiers based on the relationship between their types. For example, two identifiers with types in a 1:1 relationship indicate that one might be able to use the two identifiers to identify an object interchangeably. Two

[3]Stitch identifies the YARN application ID as interchangeable with the MapReduce job ID.

identifiers with types in an m:n relationship suggests that their combination is required to unambiguously identify an object (as discussed below). Finally, two correlated objects with IDs of types in a 1:n relationship indicate a hierarchical relationship between the objects they represent; i.e., one likely created or forked the other.

To illustrate how the relationship between object types can be useful, consider the example log snippet in Figure 3, which is the slightly simplified log output when processing two Hive queries. Messages 2-25 are output when processing query_14 submitted by Hive. YARN assigns it to an application (app_14), which in turn spawns two map and two reduce task attempts. Each attempt is dispatched and executed in a container. After the map phase, each reduce attempt creates two fetchers. Each fetcher is a thread that shuffles output from map attempts. Messages 27-33 show events related to query_15 where map and reduce attempts fail and get reassigned to different containers. It also shows that the same container can be reused by multiple attempts.

Figure 4 shows the relationships between each pair of object *types*. We call this the *Type Relation graph*. We explain a few of the relationships. First, while a user can submit multiple queries, a query is always uniquely associated with a single user; hence USER $\rightarrow$ QUERY, a 1:n relationship. Further, the application ID (e.g., 14 in "app_14") is included as part of both the identifiers of the MapReduce attempts and the containers spawned by this application; hence APP $\rightarrow$ ATTEMPT_M, APP $\rightarrow$ ATTEMPT_R, and APP $\rightarrow$ CONTAINER. Note that Stitch is able to parse map attempts as being of a different type than reduce attempts because they have different "schemas" ("attempt(.*)_m_(.*)" versus "attempt(.*)_r_(.*)"). The details of our log parser will be discussed in Section 4. ATTEMPT_R has an m:n relationship with CONTAINER because a container can be reused by multiple attempts while an attempt can also be assigned to multiple containers, given container failures.

Algorithm 1 shows how we further identify objects unambiguously given that some of the identifiers used to refer to objects are ambiguous. The algorithm takes two inputs: the Type Relation graph and the entire set of EVENTS. Each event, E, is represented as a set of object types based on the IDs found in the event log message. For example, E might be {USER, QUERY, APP}, as extracted from line 2 in Figure 3. The algorithm converts the *Type Relation graph* into a *System Stack Structure graph*, or S^3 graph, in a sequence of steps. Each node in the graph represents an object *type* along with its *signature*. This signature is the set of object identifier types defined by the requirement that an ID of each type must be present in order to unambiguously identify the object.

We start by setting the signature of every node in the Type Relation graph to the type of the object. The algorithm then goes through the following steps:

```
1  Hive user1 login successfully
2  Hive user1 submits query_14 : app=app_14
3  RM Application app_14 is submitted
4  AM app_14 created task attempt14_m_0
5  AM app_14 created task attempt14_m_5
6  AM app_14 created task attempt14_r_0
7  AM app_14 created task attempt14_r_1
8  RM app_14 allocated container14_2
9  RM app_14 allocated container14_3
10 RM app_14 allocated container14_8
11 RM app_14 allocated container14_9
12 AM Dispatch attempt14_m_0 on container14_2
13 AM Dispatch attempt14_m_5 on container14_3
14 AM Dispatch attempt14_r_0 on container14_8
15 AM Dispatch attempt14_r_1 on container14_9
16 MR container14_8 creates thread fetcher1
17 MR container14_8 creates thread fetcher2
18 MR container14_9 creates thread fetcher1
19 MR container14_9 creates thread fetcher2
20 MR container14_8 fetcher1 shuffle attempt14_m_0
21 MR container14_8 fetcher2 shuffle attempt14_m_5
22 MR container14_9 fetcher1 shuffle attempt14_m_5
23 MR container14_9 fetcher2 shuffle attempt14_m_0
24 RM app_14 finished
25 Hive Ended query_14
26 .. ..
27 Hive user1 submits query_15 : app=app_15 app_16
28 AM attempt15_m_0 failed on container15_0
29 AM Reassign attempt15_m_0 on container15_7
30 AM Dispatch attempt15_r_1 on container15_8
31 AM Dispatch attempt15_r_2 on container15_8
32 AM attempt15_r_2 failed on container15_8
33 AM Reassign attempt15_r_2 on container15_9
```

Figure 3: Log snippet output by two Hive queries. The software component that output each log message is shown at the beginning of the line. RM and AM stand for YARN's Resource Manager and Application Manager, respectively. MR stands for MapReduce. Identifiers of objects of different types are shown in different colors.

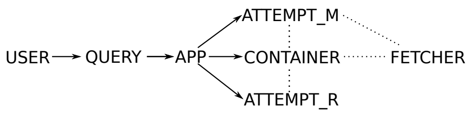


Figure 4: The Type Relation graph for the Hive log shown in Fig. 3. Each node is an object type. A solid arrow represents a 1:n relationship between the source and the destination object types; a dotted line represents an m:n relationship. There is no 1:1 relationship between types in the Hive log example.

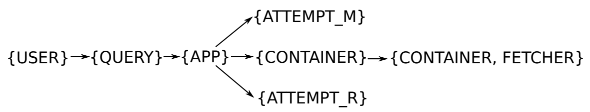


Figure 5: S^3 Graph of the Hive log snippet shown in Fig. 3.

Step 1: Merge 1:1 nodes. We first attempt to merge the nodes that are connected with $\equiv$ edges. If two types have a 1:1 relationship, then the IDs of those types may often be used interchangeably to represent the same object. However, this is not always true. For example, YARN creates a unique url for each reduce task attempt so that

Algorithm 1: S^3 graph construction

```
   Input  : G: Type Relation Graph, EVENTS
   Output: System Stack Structure ($S^3$) Graph
   /* Step 1:  merge ≡ nodes in G          */
1  foreach connected component C of ≡ relations do
2      foreach subset S: {$T_1$,..$T_n$} in decreasing size do
3          if $\forall i \in [1,n], \forall obj_i \in T_i$,
           $\exists obj_1 \in T_1, ..obj_n \in T_n$ s.t.
           $obj_1 \equiv ..obj_i \equiv ..obj_n$ then
4              Node N ← merge($T_1, T_2, .., T_n$);
5              N.sig ← hash($T_1$.sig, $T_2$.sig, .. $T_n$.sig);
6              replace $T_i$.sig with N.sig in EVENTS;
7          end
8      end
9  end
10 remove any outstanding ≡ edges;
   /* Step 2:  process ⋈ relations          */
11 foreach connected component C of ⋈ relations do
12     S ← {all the types in C};
13     foreach E ∈ EVENTS do
14         sig ← $E \cap S$;
15         if ∄ Node n where n.sig = sig then
16             nv ← new Node() with nv.sig ← sig;
17             if ∃ Node n' where n'.sig ⊂ nv.sig then
18                 add edge n' → nv;
19             end
20         else
21             mark n;
22         end
23     end
24     remove all unmarked nodes from C;
25 end
   /* Step 3:  filter non-object-types      */
26 foreach Node n do
27     if ∃Node n1,n2 s.t. n1.sig ∩ n2.sig = ∅ and
       n1.sig ∪ n2.sig ⊆ n.sig then
28         remove n;
29     end
30 end
```

a user can monitor the progress of this attempt in a web browser. Consequently, we infer ATTEMPT_R ≡ URL. However, URL is a generic type, and there can be other urls that are not related to any reduce attempt. For example, every job has its configuration information stored in an XML file that is referenced by a url. This XML file url does not appear together with any reduce attempt in any event. Therefore, we cannot say that URL and ATTEMPT_R may be used interchangeably. (Note that Stitch still infers ATTEMPT_R ≡ URL because for every pair of reduce attempt (att_i) and url (url_j) such that $\text{att}_i \sim \text{url}_j$, we have $\text{att}_i \vdash \text{url}_j$ and $\text{url}_j \vdash \text{att}_i$.)

Instead, we only merge those nodes $T_1, T_2, ..T_n$ in a ≡ connected component whose types can indeed be used interchangeably (line 3); i.e., when for any obj_i of type T_i, there exists obj_1 of type T_1, obj_2 of type T_2, ..., obj_n of type T_n such that $\text{obj}_1 \equiv \text{obj}_2.. \equiv \text{obj}_n$, where $\text{obj}_i \equiv \text{obj}_j$ iff $\text{obj}_i \sim \text{obj}_j \wedge \text{obj}_i \vdash \text{obj}_j \wedge \text{obj}_j \vdash \text{obj}_i$. This prevents ATTEMPT_R and URL from being merged because there exist urls, such as the XML file url, that are not correlated with any reduce attempt. The fact that the types of the merged nodes can be used interchangeably indicates they are redundant. To merge $\{T_1, ..T_n\}$, we hash their signatures into a single value representing a new "type", and we replace every T_i in EVENTS with this hash value. After this, the outstanding ≡ edges, such as ATTEMPT_R ≡ URL, are removed as the types that are connected by them are not truly interchangeable.

Step 2: Process m:n nodes. In order to be able to identify objects unambiguously, we consider combining types with m:n relationships. The challenge is to determine which types should be combined. For example, "HOST", "PID", and "TID" (i.e., thread ID) have an m:n relationship between each pair. While {HOST}, {HOST,PID}, and {HOST,PID,TID} are meaningful combinations as they unambiguously identify hosts, processes, and threads respectively, the combination of {HOST,TID} is meaningless. To eliminate meaningless combinations, we consider all of the different combinations the programmers output in the log statements and only include the type combinations that appear in at least one log message. The reasoning is as follows: if a combination of identifiers is necessary to represent an object unambiguously, then the programmer will always output them together. A meaningless combination, such as {HOST,TID}, will likely never be found alone in a log message without "PID", so combinations such as these are discarded.

Therefore, for each ⋈-connected component, C, we only consider the type subsets where there exists an E ∈ EVENTS represented by a log message that contains exactly the types in this subset, but not any type in its complement set (line 11-25). A node whose type always appears with other types in the same C is removed at the end (line 24).

For the Type Relation graph shown in Figure 4, Step 2 creates four new nodes: {CONTAINER, FETCHER}, {CONTAINER, FETCHER, ATTEMPT_M}, {CONTAINER, ATTEMPT_R}, and {CONTAINER, ATTEMPT_M}. After creating the nodes, we further add 1:n edges from less constrained object types to more constrained object types (line 17-19). For example, a → edge will be added from node {CONTAINER} to {CONTAINER, FETCHER}.

Step 3: Filter non-objects. One should note that not every node created in the previous step is an actual object type in the system. Among the nodes that are created in Step 2 for the Hive example, only the one whose signature is {CONTAINER, FETCHER} represents a true object type, namely a fetcher thread in a container process. To filter out non-object types, Stitch removes nodes that are a combination of two existing object types. Hence, in our example, it would remove {CONTAINER, FETCHER, ATTEMPT_M}, {CONTAINER, ATTEMPT_R} and {CONTAINER, ATTEMPT_M} because they are combinations of other object types.

Figure 5 shows the S^3 Graph constructed by Stitch from the Hive log. This graph provides a simple model of the system. Each node is a type of object, and each edge represents a 1:n relationship, which indicates a hierarchical relationship, such as fork or creation.

Note that the S^3 Graph should be cycle free, because objects do not generally have a circular fork or creation pattern. If a cycle exists, then it must already have existed in the Type Relation graph since Algorithm 1 does not introduce any cycles. Therefore, we first run a cycle detection algorithm on the Type Relation graph. If a cycle is detected, every 1:n edge in the cycle is conservatively changed to an m:n edge. In our experimental evaluation, however, we never once encountered a cycle.

Generating a graph of object instances with their interactions, like the ones shown in Section 2, from an S^3 graph is merely a pattern matching appliction. Section 4.2 describes the algorithm in detail.

4 Implementation

Stitch uses a client-server model. A client runs on all hosts being monitored to: (1) locate active logs, (2) parse every log event into a set of object identifiers, and then (3) send the events to the server. A centralized server analyzes the events from all clients to build the S^3 graph and instantiates it with object instances.

4.1 Client Implementation

The client, implemented on Linux, runs as a daemon process. The interval between the times the client wakes up is called an *epoch*. Every time it wakes up, it scans the `/proc` file system to find all running processes. The client then examines each process' file descriptors to locate log files. We treat a file as a log if its type (determined by the file's magic number) is ASCII text and its name or a parent directory's name contains the text "log".

For each process with an open log file, the client tries to locate executables of the process (including dynamically linked libraries) by searching through its file descriptors and memory mapped files. For JVM processes, Stitch also searches the process' classpath for all .jar, .war, and .class files. This ensures that executables are found even if they were already closed by the JVM. Similarly, for Python processes, the client identifies the starting script from the shell command (e.g., ./script.py) and then uses Python's ModuleFinder package to locate the remaining scripts in the dependency graph, regardless of whether they are currently open.

Next, the client extracts all constant strings from each executable. For ELF executables, we extract constants from the read-only data segments (i.e., .rodata and .rodata1) by treating "\0" as a string terminator. For Java class files we extract the strings from each file's constant pool. For Python bytecode, we extract strings from the co_consts field in the Python code object. Currently, these are the only executable formats we support.

Parsing logs. The goal of log parsing is to extract the identifier values and infer their types from each log message. If an executable's constant string contains format specifiers, then this string can be directly used as a regular expression, where the specifiers are metacharacters (e.g., "%d" can be converted to "(\d+)" to extract an integer). We treat the pattern matched by a format specifier as a variable value.

However, most variable values output to log messages from Java, Scala, C++, and Python programs use string concatenation operators; e.g., "2016-04-02T00:58:48.734 MongoDB starting : pid=22925 port=27017 dbpath=/var/lib/mongodb" is printed by the following code snippet:

```
l << "MongoDB starting : pid=" << pid
  << " port=" << serverGlobalParams.port
  << " dbpath=" << storageGlobalParams.dbpath;
```

For these output methods, we use an approach general to all of the aforementioned languages: for each log message, any segment that matches a constant string is treated as static text, leaving only the variable values. In the example, "MongoDB starting : pid=", " port=", " dbpath=" are three of the constant strings parsed from MongoDB's executable, leaving "22925", "27017", and "/var/lib/mongodb" as variable values.

Stitch solves this string matching problem with a dynamic programming algorithm. Given a log string of length n, $L[0..n-1]$, let $M(i)$ be the maximum number of characters in $L[0..i]$ that are matched by constant strings. Our goal is to find the subset of constant strings that matches $M(n-1)$ characters of L in a non-overlapping manner. To compute this, we first define a function $match()$ as the following:

$$match(a,b) = \begin{cases} b-a+1 & \text{if } L[a..b] \text{ matches a constant} \\ 0 & \text{otherwise} \end{cases}$$

Now we can iteratively compute $M(i)$:

$$M(i) = \max\{match(0,i), \max_{0\le j<i}\{M(j)+match(j+1,i)\}\}$$

This string matching is only necessary the first time a log message type is being parsed. In this example, after parsing the message, the client builds a regular expression: "MongoDB starting : pid=(\d+) port=(\d+) dbpath=(.*)". The next time if another message is printed by the same statement, Stitch can directly match it against the regular expression. We also use a heuristic to discard any string literals with fewer than three characters since the executables we evaluated often contained most permutations of all one and two character strings; using them would miscategorize identifier values as static text.

Next, Stitch infers the type of each variable. First, the client expands the variable to include characters within the word boundary delimited by whitespace. If the expansion includes static strings, then this "schema" of constant strings serves as the variable's type. For example, consider this Hadoop log message: "app_14 created task attempt14_r_0". Initially, the occurrences of "14" and "0" are recognized as variables, while "app_", " created task ", "attempt", and "_r_" are constant strings. Following expansion, the types of these two variables are "app_(\d+)" and "attempt(\d+)_r_(\d+)".

If a variable still does not include constant strings after the expansion, we trace backwards starting from the variable and use the first matched static text alphabetical word as the type. For example, in the MongoDB example, the three variables would have the types "pid", "port", and "dbpath" respectively.

Finally, the client uses a pair of heuristics to avoid capturing non-identifier variables. The first heuristic eliminates variables with types that do not end with a noun since intuitively, identifiers have noun-based types. For example, in the log, "Slow BlockReceiver write packet to mirror took 20ms", the latency variable is eliminated since the preceding static text, "took", is a verb. The next heuristic eliminates variables whose types are common non-identifiers (e.g., "size", "usage", "progress", etc.).

In practice, these heuristics did not eliminate all non-identifiers necessitating user-intervention. For example, "user" sometimes referred to a program's user-space execution time rather than a username. Rather than implementing heuristics for every such corner case, we allow the user to modify the generated regexes, which is only a one-time effort for each system.

Network protocol. At the end of each epoch, Stitch sends the parsed log messages from the last epoch to the server. The network protocol includes the following fields: (1) the timestamp of the epoch; and (2) a list of tuples, each with the format:

<severity, log file, {ID1:type1, ID2:type2, ..}, count>
All log messages from the same log file with the same set of identifiers and severity (e.g., INFO, WARN, etc.) are aggregated into a single tuple with the "count" field indicating the number of such log messages. This protocol message is then sent using Rsyslog [33] since communication is unidirectional.

The length of an epoch presents a trade-off between the timeliness of monitoring and the amount of network traffic. An epoch length of zero will force the client to stay awake and send parsed log messages one at a time; a large epoch will "compress" log messages that have the same set of identifiers within the epoch into a single tuple. Since log messages often arrive in a bursty manner, even a small epoch can significantly reduce network traffic. We set the epoch to be one second in our experimental evaluation.

4.2 Server Implementation

The Stitch server is also implemented as a daemon process. It consists of two threads: The first matches the stream of incoming events against the S^3 graph to generate an instantiated S^3 graph, henceforth called an S_i^3 graph. Each node in the S_i^3 graph is an object *instance*, whose signature is a set of identifier *values* instead of types as in the S^3 graph. The node also records the set of events that include the object instance.

For each event, e, we say e *instantiates* a node, N, from the S^3 graph if the set of identifier *types* in e is a superset of those in N's signature. For example, both events {app_14} and {app_14, attempt14_m_0} instantiate node {APP}. Initially, when no object instances have been created, for each incoming event, Stitch checks whether the event instantiates any of the *root* nodes in the S^3 graph. If so, Stitch creates an object instance node in the S_i^3 graph. For example, event {user1} will cause the creation of a node in S_i^3 graph, with signature {user1}.

Once an object instance node has been created in S_i^3 graph, for each incoming event, Stitch first checks if it matches any of the existing S_i^3 nodes. We say an event, e, matches a node, n, in the S_i^3 graph if e's identifier set is a superset of n's signature. If so, e is added to the event set of n. For each node, n, that e matches, Stitch further checks if e can instantiate any of the children of node, N, in the S^3 graph (where n is instantiated from N). If so, Stitch further instantiates the children of N and adds them as children of n. If one event matches multiple S_i^3 nodes that are not on the same path, a link is created between each node pair, indicating an interaction between them (Figure 2 in Section 2 showed an example of links represented as vertical lines between nodes).

Consider the S^3 graph for the Hive example shown in Figure 5. Figure 6 shows the S_i^3 graph Stitch generates after analyzing the first five log messages. The first message instantiates the node "user1". The second message matches the first node, instantiates a child node "query_14", and then further instantiates node "app_14"

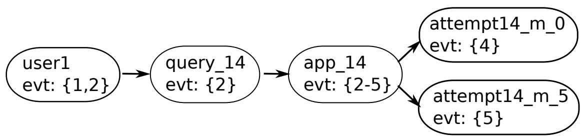


Figure 6: The S_i^3 graph generated from the first five log messages in Figure 3. Each node is an object instance. The events that include the object are also shown on each node.

Name	Software components	Benchmark
Hive	Hive, YARN, MapReduce, HDFS	HiBench
Spark	Spark, Spark SQL, Hive, HDFS	BigBench
OpenStack	Horizon, Glance, Nova, Keystone	VM cycle
Production	CRON, dbus, dhclient, dnsmasq, NetworkManager, ntpd, sshd, su, PostgreSQL	Common Linux ops

Table 1: The systems we used in evaluation.

as a child of node "query_14". The third message matches node "app_14", but does not instantiate any new nodes. The fourth and fifth messages match node "app_14" and then instantiate two nodes with signatures "attempt14_m_0" and "attempt14_m_5" respectively.

The server's second thread builds the S^3 graph, and incrementally updates it based on new patterns observed from incoming events. The thread first updates the Type Relation graph incrementally based on the observation that the relationship between two object types can only be updated in one direction: 1:1 $\rightarrow$ 1:n $\rightarrow$ m:n. Once the Type Relation graph is up to date, the server rebuilds the S^3 graph using Algorithm 1 and notifies the first thread so that it always uses the latest S^3 graph to build S_i^3 graph.

Visualization. Stitch's visualization front-end loads the S_i^3 graph as a JSON file and displays each node and its events as a row in a two-panel layout as shown in Figure 1. In order to scale to thousands of events per node, Stitch renders the graph using HTML5 Canvas and avoids drawing hidden elements where possible. For each node the user drills-down, Stitch performs a transitive reduction on the edges between this node and its children. Performing transitive reduction interactively like this avoids the overhead of doing it upfront since this is an $O(E)$ algorithm, where E is the number of edges.

5 Experimental Evaluation

We answer the following questions in evaluating Stitch: (1) How much time does Stitch save in profiling and debugging real-world systems? (2) Do real-world systems follow the Flow Reconstruction Principle? (3) How accurate is Stitch in identifying objects and their interactions? (4) What is the performance of Stitch?

We evaluated Stitch using both a controlled user study and lab experiments. The experiments consisted of using Stitch to monitor each system and workload listed in Table 1. In total there are 19 software components, programmed in C/C++, Java, Scala, and Python. The distributed system stacks were: (1) Hive stack – consisting of Hive, YARN, MapReduce, and HDFS – driven by the HiBench [15] workloads that repeatedly create, query, and delete tables; (2) Spark stack, with BigBench [12] used to send queries to Spark SQL, which then submitted them to a Spark cluster that read data from HDFS; and (3) OpenStack – consisting of four major software components – driven by our own workload generator. The generator used 80 concurrent processes to repeatedly create, boot, suspend/resume, pause/un-pause, and shutdown each VM at a randomized pace. Each distributed system workload was run for more than 24 hours on a cluster of 200 VMs.

Case	Description
OpenStack	Identify the hierarchy of components involved in creating VM instances.
MapReduce	Identify the bottleneck of a job when a reduce task was scheduled on a slow CPU.
YARN	Debug the anomaly described in Section 2.
Spark	Jobs slowed down by 500% because all but one HDFS datanode went down.
HDFS	Jobs slowed down due to a slow network link which affected HDFS read latency [29].

Table 2: Real-world profiling and debugging tasks we evaluate with Stitch. The first three are further used in our user study.

The production system was our own 24-node cluster, used for a variety of daily development workloads. Over the course of five months, Stitch identified and monitored nine log-printing components in the system, including job-scheduling, system-messaging, networking, session-management, privilege-management, and database utilities. Eight of these components logged using syslog.

5.1 User Study

Ultimately, the value of any debugging tool should be measured by the amount of time it saves. We evaluated this in a controlled user study on the first three debugging and profiling tasks listed in Table 2. They cover some real-world scenarios where Stitch can be used: (*i*) understanding the object hierarchy of a software stack ("OpenStack"); (*ii*) identifying bottlenecks to debug an anomaly we experienced in our production cluster that took us a long time to understand without Stitch ("MapReduce"); and (*iii*) debugging the anomaly described in Section 2 ("YARN"). We reproduced each case on a three nodes cluster. 14 individuals participated in the study including an experienced system administrator, professional developers, and both graduate and senior undergraduate students. All of them were experienced programmers familiar with Linux utilities such as grep, but their experience

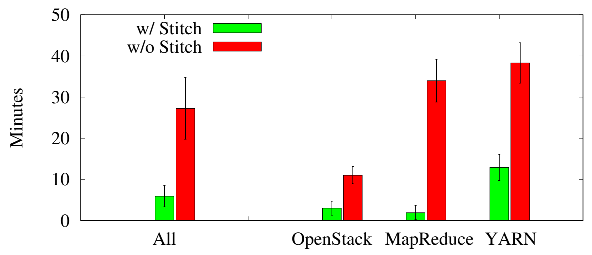


Figure 7: The result of Stitch's user study. The error bars show a 95% confidence interval.

with OpenStack and Hadoop varied, ranging from "no familiarity" to "expert developers". No co-author of this paper was a participant.

Each participant was given all three cases. Half of the participants were allowed to use Stitch for two cases but only logs (and Linux utilities) for the other case. The other half was assigned the opposite configuration. As a result, the collective expertise and experience was evenly spread across the different cases. Before the experiments, we gave each participant a five-minute verbal introduction to both OpenStack and Hadoop as well as a five-minute demo of Stitch. The demo was on a simple MapReduce job and served to familiarize users with the user interface. We then asked the participants to debug each case. Since the experiment is a single-blind trial (i.e., we, the co-authors, knew the answers), we gave the participants written instructions for each case and then minimized our interactions with them in order to avoid any potential influence. A 45 minute time limit was imposed on each case.

We marked a case as solved when the user identified the underlying cause to the degree that the raw logs allowed. A case was marked as unsolved if the user exceeded the 45 minute time limit. In the OpenStack case, success meant identifying the components involved in the creation of a VM instance. For the MapReduce case, it was to identify which job was slower and that the job was likely bottlenecked by the machine's slow CPU. For the YARN case, it was to identify that the longest query was bottlenecked waiting for a container to be released by another user's query.

Note that this is only a best effort experiment. The following potential threats to validity should be considered when interpreting the results. First, there is a potential bias in the selection of participants, as many of them were not familiar with the target systems. We note, however, that a comprehensive study [39] has shown many real-world debugging tasks are also performed by programmers unfamiliar with the target system. Another threat is that a study on just three cases may not be representative. We actually started to evaluate a fourth case ("HDFS" in Table 2) but decided to drop it after two trials (despite the initial results showing the largest time savings) because the participants became too exhausted given the complexity of debugging real-world systems without appropriate tools.

Figure 7 shows the results of user study. On average users spent 27.3 minutes without Stitch and 5.9 minutes with Stitch for a 4.6x speedup. An unpaired T-test shows that the hypothesis "Stitch lowers debugging time" is true with a probability of 99.99% (P value < 0.0001).

This measured speedup is likely an underestimate due to the 45-minute time limit we imposed. Among the 21 trials where Stitch was not provided, participants reached the time limit in eight of them without having found the issue. This suggests that debugging real-world distributed systems is indeed a complex task. In contrast, all of the Stitch users successfully diagnosed the case within the time limit.

Interestingly, participants using Stitch accidentally found an anomaly we were not targeting: when debugging the YARN case, some participants noticed that the map containers seemed to have long idle periods in the middle of their executions. It turned out that we had inserted a sleep call in map tasks so to reproduce the bug. Once we realized this, we removed the sleep call and instead used a larger input size to reproduce the anomaly.

Fundamentally, Stitch offers a "top-down" approach, whereas other practices, like "grep", are "bottom-up". While we show that Stitch is effective overall, the user study also showed that there are cases where a "bottom-up" approach is more effective. For example, when debugging the YARN case, a participant not using Stitch diagnosed the anomaly in 10 minutes because he spotted a log message clearly stating that a container could not be allocated. This suggests a desirable capability that would enable Stitch to *detect* some anomalies more quickly. While a comprehensive solution is beyond the scope of this paper, simple opportunities exist like highlighting events with "ERROR" or "FATAL" verbosity.

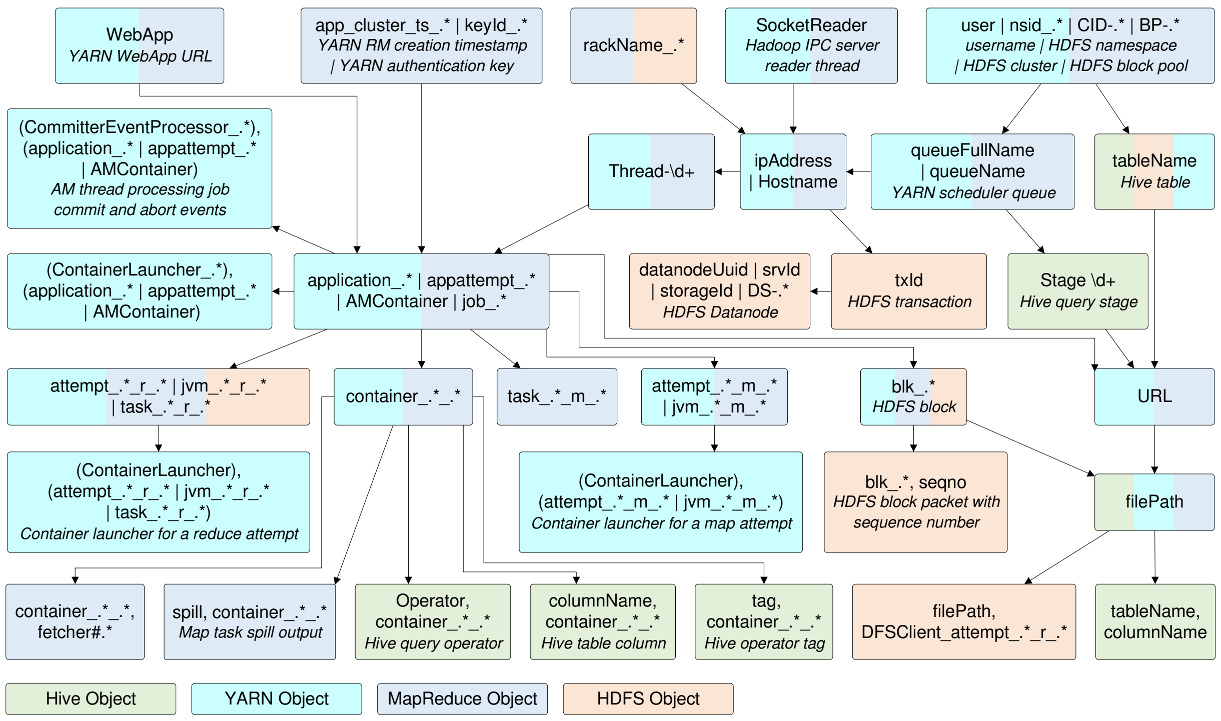
5.2 Other Real-world Cases

Table 2 shows two other real-world failures used to evaluate Stitch. The Spark failure was an anomaly we personally experienced. Stitch helped diagnose this failure by correlating objects along the path from a slow query all the way to the HDFS file read bottlenecking the query.

The HDFS case is a real-world issue [29] where the output generated at default verbosity does not provide sufficient information. At default verbosity, Stitch only allowed users to detect the fact that HDFS file access was slow and that a particular datanode was slow, but it was necessary to increase the logging level to allow Stitch to help identify a slow network link as the root cause; i.e., "DEBUG" logging was necessary to have the per packet latencies output to the log files.

	Log events		Identifiers		Objects			Edges		
	Instances	Types	Instances	Types	Instances	Types	Accuracy	Instance	Types	Accuracy
Hive	3,981,149	206	915,872	69	295,042	32	90%	401,924	73	87%
Spark	8,203,395	151	3,627,885	56	192,969	31	94%	485,133	60	93%
OpenStack	2,336,227	20	766,203	13	214,822	14	100%	2,196,315	19	100%
Production	312,779	36	123,668	22	8,141	24	100%	16,056	41	98%
Total	14,833,550	413	5,433,628	160	711,034	101	96%	3,099,428	193	95%

Table 3: Accuracy of Stitch's object reconstruction.



WebApp
YARN WebApp URL
app_cluster_ts_.* | keyId_.*
YARN RM creation timestamp | YARN authentication key
rackName_.*
SocketReader
Hadoop IPC server reader thread
user | nsid_.* | CID-.* | BP-.*
username | HDFS namespace | HDFS cluster | HDFS block pool
(CommitterEventProcessor_.*), (application_.* | appattempt_.* | AMContainer)
AM thread processing job commit and abort events
Thread-\d+
ipAddress | Hostname
queueFullName | queueName
YARN scheduler queue
tableName
Hive table
(ContainerLauncher_.*), (application_.* | appattempt_.* | AMContainer)
application_.* | appattempt_.* | AMContainer | job_.*
datanodeUuid | srvId | storageId | DS-.*
HDFS Datanode
txId
HDFS transaction
Stage \d+
Hive query stage
attempt_.*_r_.* | jvm_.*_r_.* | task_.*_r_.*
container_.*_.*
task_.*_m_.*
attempt_.*_m_.* | jvm_.*_m_.*
blk_.*
HDFS block
URL
(ContainerLauncher), (attempt_.*_r_.* | jvm_.*_r_.* | task_.*_r_.*)
Container launcher for a reduce attempt
(ContainerLauncher), (attempt_.*_m_.* | jvm_.*_m_.*)
Container launcher for a map attempt
blk_.*, seqno
HDFS block packet with sequence number
filePath
container_.*_.*, fetcher#.*
spill, container_.*_.*
Map task spill output
Operator, container_.*_.*
Hive query operator
columnName, container_.*_.*
Hive table column
tag, container_.*_.*
Hive operator tag
filePath, DFSClient_attempt_.*_r_.*
tableName, columnName
Hive Object
YARN Object
MapReduce Object
HDFS Object

Figure 8: The entire, unfiltered System Stack Structure graph for the Hive stack after applying a transitive reduction. We explain the object type in italics when necessary. A '|' in the signature indicates that the identifier types can be used interchangeably, while a ',' indicates the types must appear together to unambiguously identify an object. The colors of each node indicate the components that output its object instance.

5.3 Object Reconstruction Accuracy

Table 3 shows Stitch's accuracy at identifying objects across four different system stacks. For each workload, we used Stitch to analyze the complete set of logs without performing sampling. The total number of instances is formidable: there are 14.8 million log messages, from which Stitch extracted 5.4 million identifier values to then infer 700,000 objects and 3.1 million edges. However, Table 3 also shows Stitch's power in reducing complexity by extracting the underlying system model. There are only 413 log event types, 160 identifier types, 101 object types, and 193 edges.

Figure 8 shows the entire, unfiltered S^3 graph inferred from the 4 million messages produced by the Hive stack. It clearly shows the hierarchical structure of the system.

For each system, we evaluated the precision of Stitch's object and hierarchy identification by carefully verifying every node and edge in the S^3 graph. Overall, 96% of the objects inferred by Stitch were accurate. Among the 101 object types Stitch identified, only four are incorrect. Three are from the Hive stack while the fourth is from Spark. The first one occurred because the "Thread-ID" in the Hive stack can represent threads from two components, namely YARN NodeManager (NM) and MapReduce ApplicationManager (AM). In AM, a thread ID always appears together with the AM container ID (container is a process), and the types of the corresponding objects have an m:n relationship. However, in NM a "Thread" ID can be printed by itself without a pid. This caused Stitch to infer an object type whose signature is "Thread" by itself. While this seemingly violates the Flow Reconstruction Principle, it turns out that there is only one NM process running on each node, therefore a thread ID by itself can still unambiguously identify a thread in this process. In fact, the NM's pid is never output to a log.

The second incorrect object type, "Stage", seemingly occurred due to a violation of the Flow Reconstruction Principle for a similar reason. Each stage of a Hive query has an identifier that is unique only within the context of a query. In practice however, a stage and query identifier are never printed in the same log message. This is because the Hive shell only runs a single query at a time, meaning the stages of two queries never interleave. As a result, Stitch incorrectly infers (based on the Flow Reconstruction Principle) that a stage identifier in multiple queries represents the same object.

The third incorrect object type occurred because of the Hive/Hadoop configuration we used. In Hadoop, "Socket_Reader" is the ID of a thread within the IPC Server process. The identifier is unique only within the context of an IPC Server process. Stitch only observes a 1:n relation between thread and IPC process (instead of the correct m:n relation) because, by default, the thread pool's size is set to one thread that has a fixed thread ID, yet there are multiple IPC Server processes.

The final incorrect object type occurred because of the limited scale of our Spark workload. Stitch inferred that IP address and inode can be used interchangeably (they have 1:1 relationship). However, they should have an m:n relationship since an HDFS inode is replicated onto multiple machines and each machine can host multiple HDFS inodes. This inaccuracy occurred because the two identifiers are only ever correlated in a rarely printed ERROR log message. The small scale of our workload did not output a large enough number of such errors to allow accurate inference of this relation.

Stitch achieves 100% accuracy on OpenStack because there are only 14 objects types and identifiers are automatically logged by the underlying logging library; each message is prepended with pid, user ID, project ID, etc.

The high accuracy on the production workload is also due to the underlying logging library in addition to a longer time window for analysis. Given that the production software was not distributed, we expected to find many messages with identifiers which were ambiguous across the cluster. Interestingly, this was not the case since by default, syslog automatically appends timestamp, hostname and pid to every log message. That said, analyzing just 24 hours of logs did result in poor accuracy since many relationships were 1:n instead of m:n. We had to extend the analysis time window until the model stabilized. Even then, one edge remained incorrect after over five months: only one user was using databases, thus making the relation 1:n rather than m:n. This highlights that the required log size for Stitch depends on how long it takes the system to observe all possible workload patterns.

Do real-world systems follow Flow Reconstruction Principle? Figure 9 shows the cumulative distribution of the number of objects included in each log message of the systems we evaluated. It shows that almost all of the log messages include at least one object (which also indicates that at least one identifier is included). This suggests that real-world systems are indeed following the Flow Reconstruction Principle in terms of logging identifiers. A primary reason for such a high percentage is that the underlying systems and logging libraries automatically include some identifiers. For example, by default OpenStack's logging framework prepends each message with pid, user ID, project ID, etc. In both Hive and Spark, the container and executor IDs are included in the file path of the container and executor logs respectively.

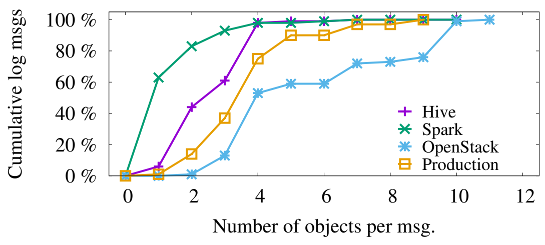


Figure 9: The cumulative distribution of the number of objects per log message.

Figure 9 also shows that over 50% of log messages contain more than one object. This suggests that objects interact in complex ways in real-world systems that further complicates debugging. This is where Stitch is particularly advantageous since it can (1) separate important fork or creation relationships from other interactions, and (2) identify which objects, at which time, interact with each other, and for how long.

We leave an evaluation of whether events are logged at critical points in the control flow as future work. However, we observe that most requests log the start and end of their execution, a clear indication of control flow.

5.4 Performance

To be viable in real-world systems, Stitch must meet two performance requirements: (*i*) it must process logs faster than they are generated; and (*ii*) it must not perturb the analyzed system's performance. We benchmarked Stitch's performance against these goals using the four previously mentioned software stacks.

Log processing. Stitch's client takes an average of 1.2 milliseconds in each one-second epoch to parse and send incoming log messages to the server. On average, the server takes 3.9 hours to process every 24 hours of log messages. Thus, both daemons are able to handle the rate of incoming log messages.

Performance perturbation. The client negligibly affects the workloads it monitors since it is CPU-bound.

Therefore, for I/O-bound jobs like the Hive workload, the client has no effect on execution time. The Spark workload can reach 100% CPU utilisation but this tends to happen in bursts. So again, the client has no effect on execution time. However, Stitch must manage the overhead of transmitting log messages over the network.

On average, Stitch achieves a 15% compression ratio compared to trasferring the complete log files for a workload. We automatically achieve 50% of this since we only need to transmit the identifiers from each log message. The other 35% comes from using Rsyslog's zlib compression on each of our protocol messages.

6 Limitation and Discussion

The efficacy of Stitch fundamentally depends on the quality of the logs. If there are no logs, or the log messages do not contain the right identifiers, then Stitch will not be able to accurately reconstruct the system model it needs. However, in that case, the logging would not even be useful for manual inspection by the developers who wrote the software. Perhaps our biggest (and happy) surprise from this project is the high quality and usefulness of the real-world logs generated by the mature distributed systems we evaluated. Nevertheless, we have also encountered real-world logs that did not contain sufficient information, as for example at one company where we tried to use Stitch (where log quality needs to be improved by enforcing the Flow Reconstruction Principle).

Another limitation of Stitch is that it cannot accurately infer causal dependencies. Instead, it can only identify correlations between objects. According to counterfactual theories of causation [19], event B causally depends on event A iff *an event B would not have occurred if event A had not occurred*. For Stitch, a causal dependence between events can be captured by two mechanisms. First, the 1:n relationships between objects can capture an object B that is created by an object A, so that their output events will have a causal dependency. Secondly, the events that are output along the same execution path involving the same object (i.e., a thread) are also captured since Stitch associates these events with that object. However, it is not true that every 1:n edge in Stitch's output captures object creation, or that every two events belonging to the same object have a control-flow dependency. We leave the job of distinguishing whether the correlation is causal or not to Stitch's user.

This limitation presents a fundamental trade-off in our identifier-only design since we ignore static text in log messages. For example, an event involving an operation on file B is causally dependent on an event involving process A if the latter event is "Process A created file B", but not if the event is only "Process A read file B". But this is something Stitch cannot identify. To capture this type of causal dependence, Stitch would need to either understand message semantics using Natural Language Processing or leverage a static analysis approach such as lprof [42]. [4]

Stitch's efficacy is also sensitive to the accuracy of extracting object identifiers and their types. In practice, it is trivial for developers or administrators to annotate the type of each identifier and its regular expression. In fact, projects like LogStash [23] already provide a database containing hundreds of regular expressions and their corresponding object types for commonly used identifiers.

7 Related Work

The fundamental challenge faced by any performance analysis tool for distributed systems is to capture the semantics of the event sequences output by the system. Existing solutions fall into one of three categories: (*i*) instrumenting the target system with predefined event semantics, (*ii*) use of static analysis to infer the system model, or (*iii*) the use of machine learning. We discuss each category in detail below, but note that Stitch does not rely on any of these techniques. Instead, it relies on programmers' intuition in event logging and builds a simple algorithm entirely based on identifiers from unstructured logs.

Intrusive approaches. Most of the existing tools capable of analyzing distributed system performance rely on instrumenting the target system [1, 2, 7, 8, 9, 10, 11, 13, 14, 24, 26, 27, 28, 32, 34]. The key benefit afforded by instrumentation is that the semantics of each event is defined by the analysis tool; therefore, the events can be unambiguously interpreted. For example, *ÜberTrace* [9] instruments Facebook's system with events that include a unique request ID and a predefined event name. Pivot Tracing [24] compiles a user's query into code that gets dynamically patched into the running system to collect tracing events; furthermore, critical identifier information is propagated to the different software components to be included in every event by the tracing code.

Stitch faces a different challenge, namely how to understand unstructured log events without any knowledge of how they are generated. This challenge presents interesting trade-offs between Stitch and the aforementioned intrusive approaches. On the one hand, because the semantics of events can be precisely interpreted, intrusive approaches can conduct more precise analyses. For example, Mystery Machine [9] and Pivot Tracing [24] can infer the *causal relationship* between events, a limitation

[4] However, it is debatable whether a log message "Process A created file B" really implies a causal relationship between process A and file B as it is possible that file B would have been created by other processes, regardless. Therefore, rigorously inferring causal dependencies between events may not be possible without domain expertise.

of Stitch we discussed in Section 6. On the other hand, the necessity of instrumentation imposes significant deployment hurdles. First, instrumentations are often specific to a particular system and cannot be generally applied to another. For example, Pivot Tracing's instrumentation requires dynamically patching code into system-specific data structures. This challenge is perhaps why, to the best of our knowledge, Stitch is the first system that has been evaluated on heterogeneous systems implemented in different languages. Second, vendors are often reluctant to modify their production systems in the fear that the instrumentations may open up reliability vulnerabilities and add performance overhead. Finally, it simply may not be possible to instrument legacy software or third-party components.

Log analysis guided by static analysis. Tools like lprof [42] and SherLog [41] are able to analyze a system's source and bytecode to infer how the processing of a request outputs log messages. For example, via control-flow analysis, lprof is able to infer that log messages are causally related if they are on the same execution path; its data-flow analysis can infer which variables are unchanged between two log printing statements, which can be used to group the logs. However, these approaches cannot be used to correlate events across heterogeneous software components because static analysis is limited to a single software component. In addition, static analysis needs to be customized for different languages (lprof only worked on Java bytecode), requires the source for all third-party libraries, and for languages like C and C++ where non-standard dialects are prevalent, it is challenging to even get the static analysis tool to compile the code [4].

Log analysis with machine learning. Several tools use machine learning on log files to detect anomalies [3, 25, 38, 40]. These tools first learn a model from logs output by correct runs, and then apply the model on logs from failure runs. For example, CloudSeer learns an automata model from clean logs that can then be used to detect anomalies [40]. Xu *et al.,* uses Principal Component Analysis (PCA) to detect unusual patterns in the logs [38]. Stitch is complementary to these tools: these tools' goals are to detect anomalies, while Stitch's goal is to profile entire systems by analyzing the performance behavior and relationships between every object in the system. Furthermore, the machine learning approaches could be used on Stitch's graphs to identify anomalies.

Log visualization tools. In recent years, most tools in this area (e.g., Splunk [36], VMware LogInsight [22], Kibana [17], etc.) have focused on system statistics and monitoring rather than breaking down the system model like Stitch. However, tools similar to Stitch exist in the area of file systems and web browsers. InProv [6] gives the user a top-down view of file system provenance data, allowing them to click through the hierarchy of interactions between processes and files over time. Chrome's Developer Tools allow a user to zoom in to and out of the hierarchy of objects on a web page. InProv, Chrome, and Stitch highlight the importance and usefulness of browsing hierarchy when debugging.

8 Concluding Remarks

This paper presented Stitch, the first system capable of reconstructing, in a non-intrusive manner, the end-to-end execution flow of requests being serviced by distributed server stacks. Without any system specific knowledge, Stitch is able to analyze unstructured log output from heterogeneous software components and construct a system model which captures the objects involved, their lifetimes and their hierarchical relationships.

At its core, Stitch is enabled by the observation that programmers follow certain principles so that they can reliably reconstruct the executions a posteriori. These principles, which we collectively refer to as the *Flow Reconstruction Principle*, state that a sufficient number of events need to be logged at critical points in the control path and that each event log output must include a sufficient number of object identifier values to be able to disambiguate the concurrent and homogeneous executions. While these principles may seem straightforward in hindsight, discovering them was challenging and took us over two years to identify and refine.

Acknowledgements

We greatly appreciate the insightful feedback from the anonymous reviewers and our shepherd, Margo Seltzer. We thank Xiang Ren, Yongle Zhang, David Lion, Serhei Makarov, and Abhishek Rudra for the useful and thought-stimulating discussions. We also thank all of the user study participants for their time spent debugging. The idea that identifiers can be parsed by excluding static text came from a discussion with David Lie. This research is supported by an NSERC Discovery grant, a NetApp Faculty Fellowship, and a Huawei grant.

References

[1] M. K. Aguilera, J. C. Mogul, J. L. Wiener, P. Reynolds, and A. Muthitacharoen. Performance debugging for distributed systems of black boxes. In *Proc. of the 19th ACM Symposium on Operating Systems Principles*, SOSP '03, pages 74–89. ACM, 2003.

[2] P. Barham, A. Donnelly, R. Isaacs, and R. Mortier. Using magpie for request extraction and workload modelling. In *Proceedings of the 6th Conference on Symposium on*

Opearting Systems Design & Implementation, OSDI '04, pages 259–272. USENIX Association, 2004.

[3] I. Beschastnikh, Y. Brun, S. Schneider, M. Sloan, and M. D. Ernst. Leveraging existing instrumentation to automatically infer invariant-constrained models. In *Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering*, ESEC/FSE '11, pages 267–277. ACM, 2011.

[4] A. Bessey, K. Block, B. Chelf, A. Chou, B. Fulton, S. Hallem, C. Henri-Gros, A. Kamsky, S. McPeak, and D. Engler. A few billion lines of code later: Using static analysis to find bugs in the real world. *Commun. ACM*, 53(2):66–75, Feb. 2010.

[5] Chapter 1. Boost.Log v2 - 1.61.0. `http://www.boost.org/doc/libs/1_61_0/libs/log/doc/html/index.html`.

[6] M. A. Borkin, C. S. Yeh, M. Boyd, P. Macko, K. Z. Gajos, M. Seltzer, and H. Pfister. Evaluation of filesystem provenance visualization tools. *IEEE Transactions on Visualization and Computer Graphics*, 19(12):2476–2485, Dec. 2013.

[7] A. Chanda, A. L. Cox, and W. Zwaenepoel. Whodunit: Transactional profiling for multi-tier applications. In *Proceedings of the 2nd ACM SIGOPS/EuroSys European Conference on Computer Systems 2007*, EuroSys '07, pages 17–30. ACM, 2007.

[8] M. Y. Chen, E. Kiciman, E. Fratkin, A. Fox, and E. Brewer. Pinpoint: Problem determination in large, dynamic internet services. In *Proceedings of the 2002 International Conference on Dependable Systems and Networks*, DSN '02, pages 595–604. IEEE Computer Society, 2002.

[9] M. Chow, D. Meisner, J. Flinn, D. Peek, and T. F. Wenisch. The mystery machine: End-to-end performance analysis of large-scale internet services. In *Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation*, OSDI '14, pages 217–231. USENIX Association, 2014.

[10] C. Curtsinger and E. D. Berger. Coz: Finding code that counts with causal profiling. In *Proceedings of the 25th Symposium on Operating Systems Principles*, SOSP '15, pages 184–197. ACM, 2015.

[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 & Implementation*, NSDI '07, pages 271–284. USENIX Association, 2007.

[12] 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. ACM, 2013.

[13] S. L. Graham, P. B. Kessler, and M. K. Mckusick. Gprof: A call graph execution profiler. In *Proceedings of the 1982 SIGPLAN Symposium on Compiler Construction*, SIGPLAN '82, pages 120–126. ACM, 1982.

[14] Z. Guo, D. Zhou, H. Lin, M. Yang, F. Long, C. Deng, C. Liu, and L. Zhou. G2: A graph processing system for diagnosing distributed systems. In *Proceedings of the 2011 USENIX Conference on USENIX Annual Technical Conference*, USENIX ATC '11, pages 299–312. USENIX Association, 2011.

[15] S. Huang, J. Huang, J. Dai, T. Xie, and B. Huang. The HiBench benchmark suite: Characterization of the MapReduce-based data analysis. In *26th International Conference on Data Engineering Workshops*, ICDEW '10, pages 41–51. IEEE Computer Society, 2010.

[16] java.util.logging (Java Platform SE 8). `https://docs.oracle.com/javase/8/docs/api/java/util/logging/package-summary.html`.

[17] Kibana: Explore, visualize, discover data. `https://www.elastic.co/products/kibana`.

[18] P. Leach, M. Mealling, and R. Salz. A Universally Unique IDentifier (UUID) URN Namespace. RFC 4122, July 2005.

[19] D. L. Lewis. *Counterfactuals*. Blackwell Publishers, 1973.

[20] Log for C++ Project. `http://log4cpp.sourceforge.net/`.

[21] Log4j - Log4j 2 Guide - Apache Log4j 2. `http://logging.apache.org/log4j/2.x/`.

[22] VMware vCenter Log Insight: Log management and analytics. `http://www.vmware.com/ca/en/products/vcenter-log-insight`.

[23] Logstash – normalizing varying schema. `https://www.elastic.co/products/logstash`.

[24] J. Mace, R. Roelke, and R. Fonseca. Pivot tracing: Dynamic causal monitoring for distributed systems. In *Proceedings of the 25th Symposium on Operating Systems Principles*, SOSP '15, pages 378–393. ACM, 2015.

[25] K. Nagaraj, C. Killian, and J. Neville. Structured comparative analysis of systems logs to diagnose performance problems. In *Proceedings of the 9th USENIX Conference on Networked Systems Design and Implementation*, NSDI '12, pages 353–366. USENIX Association, 2012.

[26] Nagios: the industry standard in IT infrastructure monitoring. `http://www.nagios.org/`.

[27] NewRelic: Application performance management and monitoring. `http://newrelic.com/`.

[28] OProf - A system profiler for Linux. `http://oprofile.sourceforge.net/`.

[29] Poor HDFS performances: Slow BlockReceiver write packet to mirror. `http://stackoverflow.com/questions/27984331`.

[30] Section 15.7 logging - Logging facility for Python - Python 2.7.12 documentation. `https://docs.python.org/2/library/logging.html`.

[31] JIRA: YARN bug 4610. `https://issues.apache.org/jira/browse/YARN-4610`.

[32] P. Reynolds, C. Killian, J. L. Wiener, J. C. Mogul, M. A. Shah, and A. Vahdat. Pip: Detecting the unexpected in distributed systems. In *Proceedings of the 3rd Conference on Networked Systems Design & Implementation*, NSDI '06, pages 115–128. USENIX Association, 2006.

[33] RSYSLOG: the rocket-fast system for log processing. `www.rsyslog.com`.

[34] 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.

[35] SLF4J. `http://www.slf4j.org/`.

[36] Splunk log management. `http://www.splunk.com/view/log-management/SP-CAAAC6F`.

[37] The syslog protocol. `http://tools.ietf.org/html/rfc5424`.

[38] W. Xu, L. Huang, A. Fox, D. Patterson, and M. I. Jordan. Detecting large-scale system problems by mining console logs. In *Proc. of the ACM SIGOPS 22nd Symposium on Operating Systems Principles*, SOSP '09, pages 117–132. ACM, 2009.

[39] Z. Yin, D. Yuan, Y. Zhou, S. Pasupathy, and L. Bairavasundaram. How do fixes become bugs? In *Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering*, ESEC/FSE '11, pages 26–36. ACM, 2011.

[40] X. Yu, P. Joshi, J. Xu, G. Jin, H. Zhang, and G. Jiang. Cloudseer: Workflow monitoring of cloud infrastructures via interleaved logs. In *Proceedings of the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '16, pages 489–502. ACM, 2016.

[41] D. Yuan, H. Mai, W. Xiong, L. Tan, Y. Zhou, and S. Pasupathy. SherLog: error diagnosis by connecting clues from run-time logs. In *Proceedings of the 15th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '10, pages 143–154. ACM, 2010.

[42] X. Zhao, Y. Zhang, D. Lion, M. FaizanUllah, Y. Luo, D. Yuan, and M. Stumm. lprof: A non-intrusive request flow profiler for distributed systems. In *Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation*, OSDI '14, pages 629–644. USENIX Association, 2014.

Early Detection of Configuration Errors to Reduce Failure Damage

Tianyin Xu, Xinxin Jin, Peng Huang, Yuanyuan Zhou, Shan Lu*, Long Jin, Shankar Pasupathy†
University of California, San Diego *University of Chicago †NetApp, Inc.

Abstract

Early detection is the key to minimizing failure damage induced by configuration errors, especially those errors in configurations that control failure handling and fault tolerance. Since such configurations are not needed for initialization, many systems do not check their settings early (e.g., at startup time). Consequently, the errors become *latent* until their manifestations cause severe damage, such as breaking the failure handling. Such latent errors are likely to escape from sysadmins' observation and testing, and be deployed to production at scale.

Our study shows that many of today's mature, widely-used software systems are subject to latent configuration errors (referred to as LC errors) in their critically important configurations—those related to the system's reliability, availability, and serviceability. One root cause is that many (14.0%–93.2%) of these configurations do not have any special code for checking the correctness of their settings at the system's initialization time.

To help software systems detect LC errors early, we present a tool named PCHECK that analyzes the source code and automatically generates configuration checking code (called *checkers*). The checkers emulate the late execution that uses configuration values, and detect LC errors if the error manifestations are captured during the emulated execution. Our results show that PCHECK can help systems detect 75+% of real-world LC errors at the initialization phase, including 37 new LC errors that have not been exposed before. Compared with existing detection tools, it can detect 31% more LC errors.

1 Introduction

1.1 Motivation

Failures are a fact of life in today's large-scale, rapid-changing systems in cloud and data centers [7,24,30,58]. To mitigate the impact of failures, tolerance and recovery mechanisms have been widely adopted, such as employing data and node redundancy, as well as supporting fast rebooting and rollback. While these mechanisms are successful in handling individual machine failures (e.g., hardware faults and memory bugs), they are less effective in handling configuration errors [20, 24, 28], especially the errors in configurations that control the failure handling itself. For example, an erroneous fail-over configuration resulted in a 2.5 hour outage of Google App Engine in 2010, affecting millions of end users [44]. Moreover, very often, the same configuration error is deployed onto thousands of nodes and resides in persistent files on each node, making it hard to tolerate by redundancy or server rebooting. As a result, configuration errors have become one of the major causes of failures in large-scale cloud and Internet systems, as reported by many system vendors [21,34,55] and service providers [7,24,28,43].

Since it is hard to completely avoid configuration errors (after all, everyone makes mistakes; as do system administrators), similar to fatal diseases like cancer, a more practical approach is to detect such errors as early as possible in order to minimize their failure damage:

- Early detection before configuration roll-out can prevent the same error from being replicated to thousands of nodes, especially in the data-center environment.
- Unlike software bugs, configuration errors, once detected, can be fixed by sysadmins themselves with no need to go through developers. Therefore, if detected earlier, the errors can be corrected immediately before the configurations are put online for production.
- For many configurations that control the system's failure handling, early detection of errors in their settings can prevent the system from entering an unrecoverable state (before any failures happen). Often, the combination of multiple errors (e.g., a configuration error plus a software bug) can bring down the entire service, as shown in many newsworthy outages [9,42,43,45].

Unlike software bugs that typically go through various kinds of testing before releases (such as unit testing, regression testing, stress testing, system testing, etc.), system administrators often do not perform extensive testing on configurations before rolling them out to other nodes and putting the systems online [25]. Besides the lack of skills [25] and the temptation of convenience [24], the more fundamental reason is that system administrators do not have the same level of understanding on *how* and *when* the system uses each configuration value internally. Thus, they are limited to simple black-box testing such as starting the system and applying a few small workloads to see how the system behaves. Due to time and knowledge limitations, system administrators typically do not

Severity level	Latent	Non-latent
All cases	47.6%	52.4%
High severity	75.0%	25.0%

Table 1: Severity of latent versus non-latent errors among the customers' configuration issues of COMP-A. LC errors contribute to 75% of the high-severity configuration issues.

Error class	Mean	Median
Latent	1.14	1.70
Non-latent	0.87	0.41

Table 2: Diagnosis time of latent versus non-latent errors among customers' configuration issues of COMP-A. The time is normalized by the average time of all the reported issues.

perform a comprehensive suite of test cases against configuration settings, especially for those hard-to-test ones (e.g., failure/error-handling related configurations) that may require complex setups and even fault injections.

Therefore, early detection should inevitably fall onto the shoulder of the system itself—the system should automatically check as many configurations as possible at its early stages (the startup time). Unfortunately, many of today's systems either skip the checking or only check configurations right before the configuration values are used, as shown in our study (§2). Typically, at the startup time, only those configuration parameters needed for initialization are checked (or directly used), while many other parameters' checking is delayed much later until when they are used in special tasks. Since such configuration parameters are neither used nor checked during normal operations, errors in their settings go undetected until their late manifestation, e.g., under circumstances like error handling and fail-over. For simplicity, we refer to such errors as *latent configuration* (LC) errors.

LC errors can result in severe failures, as they are often associated with configurations used to control critical situations such as fail-over [44], error handling [42], backup [37], load balancing [9], mirroring [45], etc. As explained above, their detection or exposure is often too late to limit the failure damage. Take a real-world case as an example (c.f., §2: Figure 3a), an LC error in the fail-over configuration settings is detected only when the system encounters a failure (e.g., due to hardware faults or software bugs) and tries to fail-over to another component. In this case, the fail-over attempt also fails, making the entire system unavailable to all the clients.

Tables 1 and 2 compare the severity level and diagnosis time of real-world configuration issues caused by LC errors versus non-latent configuration errors (detected at the system's startup time) of COMP-A[1], a major storage company in the US. Although there have been fewer LC errors than non-latent ones, LC errors contribute to 75% of the high-severity issues and take much longer to diagnose, indicating their high impact and damage.

[1]We are required to keep the company and its products anonymous.

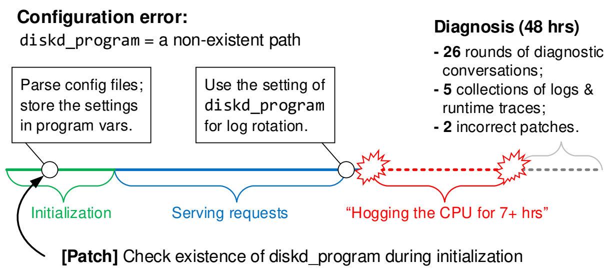
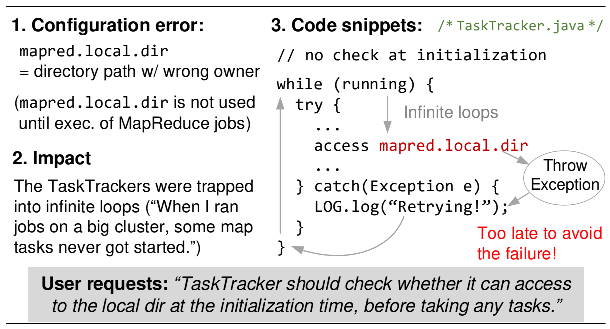

Configuration error:
diskd_program = a non-existent path
Parse config files; store the settings in program vars.
Use the setting of diskd_program for log rotation.
Diagnosis (48 hrs)
- 26 rounds of diagnostic conversations;
- 5 collections of logs & runtime traces;
- 2 incorrect patches.
Initialization
Serving requests
"Hogging the CPU for 7+ hrs"
[Patch] Check existence of diskd_program during initialization

Figure 1: A real-world LC error from Squid [37]. The error caused system hanging for 7+ hours, and resulted in 48 hours of diagnosis efforts. Later, a patch was added to check the existence of the configured path during initialization. Unfortunately, the patched check is still subject to LC errors such as incorrect file types and permissions.



1. Configuration error:
mapred.local.dir
= directory path w/ wrong owner
(mapred.local.dir is not used until exec. of MapReduce jobs)

2. Impact
The TaskTrackers were trapped into infinite loops ("When I ran jobs on a big cluster, some map tasks never got started.")

3. Code snippets: /* TaskTracker.java */

```
// no check at initialization
while (running) {
  try {
    ...
    access mapred.local.dir
    ...
  } catch(Exception e) {
    LOG.log("Retrying!");
  }
}
```

User requests: *"TaskTracker should check whether it can access to the local dir at the initialization time, before taking any tasks."*

Figure 2: A real-world LC error from MapReduce [12]. When the exception handler caught the runtime exception induced by the LC error, it was already too late to avoid the downtime. After this incident, the user requested to check the configuration "at the initialization time."

Figure 1 shows a real-world LC error from Squid, a widely used open-source Web proxy server. The LC error resided in `diskd_program`, a configuration parameter used only during log rotation. Squid did not check the configuration during initialization; thus, this error was exposed much later after days of execution. It caused 7+ hours of system downtime and cost 48 hours of diagnosis efforts. After the error was finally discerned, the Squid developers added a patch to proactively check the setting at system startup time to prevent such latent failures.

Figure 2 shows another real-world example in which an LC error failed a large-scale MapReduce job processing. This LC error was replicated to multiple nodes and crashed the TaskTrackers on those nodes. Specifically, the error caused a runtime exception on each node. The TaskTracker caught the exception and restarted the job. Unfortunately, as the error is persistent in the configuration file, restarting the job failed to get rid of the error but induced infinite loops. Note that when the exception handler caught the error, it was already too late to avoid downtime (the best choice is to terminate the jobs).

Preventing above LC-error issues would require software systems to check configurations *early* during the initialization time, even though the configuration values are only needed in much later execution or during special circumstances. This is indeed demonstrated by the developers' postmortem patches. As revealed in Facebook's recent study [43], 42% of the configuration errors that caused high-impact incidents are "obvious" errors (e.g., typos), indicating the limitations of code review and system testing in preventing LC errors. These errors might be detected by early checks (only if developers are willing to and remember to write the checking code).

1.2 State of the Art

Most prior work on handling configuration errors focuses on troubleshooting and diagnosis [5, 6, 32, 46, 48, 49, 52, 53, 56, 60, 61, 62]. The techniques proposed in these work are helpful for system administrators to identify the failure root causes faster to shorten the repair time. However, they cannot prevent failures and downtime.

Most of the existing detection tools check configuration settings against apriori correctness rules (known as *constraints*). However, as large software systems usually have hundreds to thousands of configuration parameters, it is time-consuming and error-prone to ask developers to *manually* specify every single constraint, not to mention that constraints change with software evolution [61].

So far, only a few automatic configuration-error detection tools have been proposed. Most of them detect errors by *learning* the "normal" values from large collections of configuration settings in the field [29, 36, 57, 59]. While these techniques are effective in certain scenarios, they have the following limitations, especially when being applied to cloud and data centers.

First, most of these works require a large collection of *independent* configuration settings from hundreds of machines. This is a rather strong requirement, as most cloud and data centers typically propagate the same configurations from one node to all the other nodes. Thereby, the settings from these nodes are not independent, and thus not useful for "learning". Second, they do not work well with configurations that are inherently different from one system to another (e.g., domain names, file paths, IP addresses) or incorrect settings that fall in normal ranges. They also cannot differentiate customized settings from erroneous ones. Furthermore, most of these tools target on specific error types (encoded by their predefined constraint templates) and are hard to generalize to detect other types of errors. A recent work learns constraints from KB (Knowledge Base) articles [31]. However, this approach has the same limitations discussed above. Specially, KB articles are mainly served for postmortem diagnosis and thus may not cover every single constraint.

There are very few configuration-error detection approaches that do not rely on constraints specified manually by developers or learned from large collections of independent settings (or KB articles). The only exception (to the best of our knowledge) is `conf_spellchecker` [35] which detects simple errors based on type inference from source code. While this technique is very practical, it is limited in the types of configuration errors that can be detected, as shown in our experimental evaluation (§4).

1.3 Our Contributions

This paper makes two main contributions. First, to understand the root causes and characteristics of latent configuration (LC) errors, we study the practices of configuration checking in six mature, widely-deployed software systems (HDFS, YARN, HBase, Apache, MySQL, and Squid). Our study reveals: (1) In today's software systems, many (14.0%–93.2%) of the critically important configuration parameters (those related to the system's reliability, availability, and serviceability) do not have any special code for checking the correctness of their settings. Instead, the correctness is verified (*implicitly*) when the configuration values are being actually used in operations such as a file `open` call. (2) Many (12.0%–38.6%) of these configuration parameters are not used at all during system initialization. (3) Resulting from (1) and (2), 4.7%–38.6% of these critically important configuration parameters do not have any early checks and are thereby subject to LC errors that can cause severe impact on the system's dependability.

Second, to help systems detect LC errors early, we present a tool named PCHECK that analyzes the source code and automatically generates configuration checking code (called *checkers*) to validate the system's configuration settings at the initialization phase. PCHECK takes a unique and intuitive method to check each configuration setting—*emulating the late execution that uses the configuration value; meanwhile capturing any anomalies exposed during the execution as the evidence of configuration errors*. PCHECK does not require developers to manually implement checking logic, nor rely on learning a large volume of configuration data. The checkers generated by PCHECK are *generic*: they are not limited to any specific, predefined rule patterns, but are derived from how the program uses the parameters.

PCHECK shows that it is feasible to *accurately* and *safely* emulate late execution that uses configurations. It statically extracts the instructions that transform, propagate, and use the configuration values from the system program. To execute these instructions, PCHECK makes a best effort to produce the necessary execution context (values of dependent variables) that can be determined statically. PCHECK also "sandboxes" the emulated execution by instruction rewriting to prevent side effects on the running system or its environment.

More importantly, emulating the execution can expose many configuration errors as runtime anomalies (e.g., exceptions and error code) and the emulated execution runs in a short period. PCHECK inserts instructions to capture the anomalies that may occur during the emulated execution, as the evidence to report configuration errors.

As an enforcement, PCHECK encapsulates the emulated execution and error capturing code into checkers for every configuration parameter, and invokes the checkers at the system's initialization phase. This can minimize potential LC errors, and compensate for the missing and incomplete configuration checks in real-world systems.

We implement PCHECK for C and Java programs on top of the LLVM [4] and Soot [3] compiler frameworks. We apply PCHECK to 58 real-world LC errors of various error types occurred in widely-used systems (each leads to severe failure damage), including 37 new LC errors that have not been exposed before. Our results show that PCHECK can detect 75+% of these real-world LC errors at the system's startup time. Compared with the existing detection tools, it can detect 31% more LC errors.

2 Understanding Root Causes of Latent Configuration Errors

To understand the root causes and characteristics of LC errors, we study the practices of the configuration checking and error detection in six mature, widely-deployed open-source software systems (c.f., Table 3). They cover multiple functionalities and languages, and include both single-machine and distributed systems.

We focus on configuration parameters used in components related to the system's Reliability, Availability, and Serviceability (known as RAS for short [50]). For each system considered, we select all the configuration parameters of RAS-related features based on the software's official documents, including error handling, fail-over, data backup, recovery, error logging and notification, etc. The last column of Table 3 shows the number of the studied RAS parameters. Compared with configurations of other system components, configurations used by RAS components are more likely to be subject to LC errors due to their inherently latent nature; moreover, the impact of errors in RAS configurations is usually more severe.

Note: LC errors are not limited to RAS components. Thus, the reported numbers may not represent the overall statistics of all the LC errors in the studied systems. In addition, PCHECK, the tool presented in §3, applies to all the configuration parameters; it does not require manual efforts to select out RAS parameters.

2.1 Methodology

We manually inspect the source code related to RAS configuration parameters of the studied systems. First, for each RAS parameter, we study the code that checks the parameter setting at the system's initialization phase[2] (if any) and the code that later uses the parameter's value. Then, we compare these two sets of code (checking versus usage) and examine if the initial checking is sufficient to detect configuration errors. If an error can escape from the initialization phase and break the usage code, it is a potential LC error.

We verify each LC error discovered from source code by exposing and observing the impact of the error. We first inject the errors into the system's configuration files and launch the system; then we trigger the manifestation conditions to expose the error impact. For example, to verify the LC errors in the HDFS auto-failover feature, we start HDFS with the erroneous fail-over settings, trigger the fail-over procedure by killing the active NameNode, and examine if the fail-over can succeed. As all the LC errors are verified through their manifestation, there is no false positive in the reported numbers.

Software	Description	Lang.	# Parameters	
			Total	RAS
HDFS	Dist. filesystem	Java	164	44
YARN	Data processing	Java	116	35
HBase	Distributed DB	Java	125	25
Apache	Web server	C	97	14
Squid	Proxy server	C/C++	216	21
MySQL	DB server	C++	462	43

Table 3: The systems and the RAS parameters studied in §2.

Software	Deficiency of initial checking		Studied param.
	Missing	Incomplete	
HDFS	41 (93.2%)	3 (6.9%)	44
YARN	29 (82.9%)	5 (14.3%)	35
HBase	18 (72.0%)	5 (2.0%)	25
Apache	4 (28.6%)	2 (14.3%)	14
Squid	9 (42.9%)	4 (19.0%)	21
MySQL	6 (14.0%)	6 (14.0%)	43

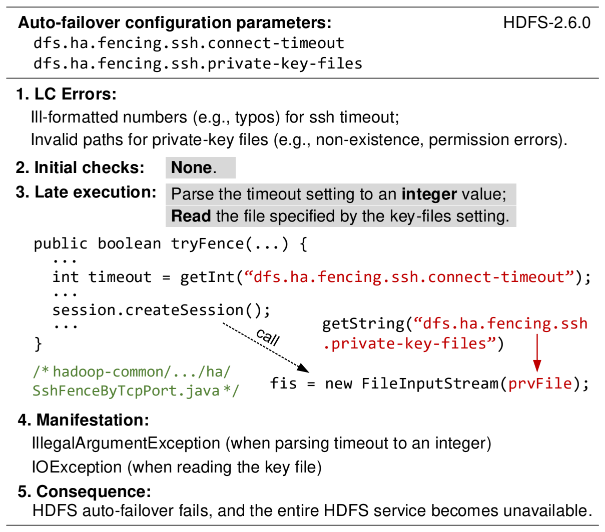
Table 4: Number of configuration parameters that do not have any initial checking code ("missing") and that only have partial checking and thus cannot detect all potential errors ("incomplete").

2.2 Findings

Finding 1: *Many (14.0%–93.2%) of the studied RAS parameters do not have any special code for checking the correctness of their settings. Instead, the correctness is verified (implicitly) when the parameters' values are actually used in operations such as a file open call.*

Table 4 shows the number of the studied RAS parameters that rely on the usage code for verifying correctness, because their initial checks are either *missing* or *incomplete*. Most of the studied RAS parameters in HDFS, YARN, and HBase do not have any special code for checking the correctness of their settings. These systems

[2] A system's initialization phase is defined from its entry point to the point it starts to serve user requests or workloads.



Auto-failover configuration parameters: HDFS-2.6.0

```
dfs.ha.fencing.ssh.connect-timeout
dfs.ha.fencing.ssh.private-key-files
```

1. LC Errors:
Ill-formatted numbers (e.g., typos) for ssh timeout;
Invalid paths for private-key files (e.g., non-existence, permission errors).

2. Initial checks: None.

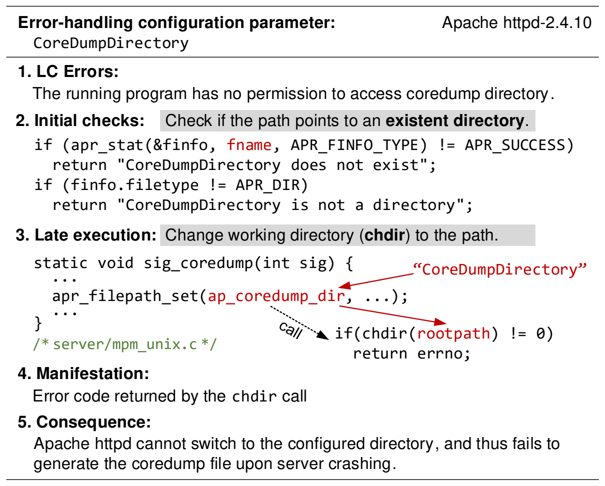
3. Late execution: Parse the timeout setting to an **integer** value; **Read** the file specified by the key-files setting.

```
public boolean tryFence(...) {
  ...
  int timeout = getInt("dfs.ha.fencing.ssh.connect-timeout");
  ...
  session.createSession();
  ...
}
/* hadoop-common/.../ha/
SshFenceByTcpPort.java */

getString("dfs.ha.fencing.ssh
.private-key-files")

fis = new FileInputStream(prvFile);
```

4. Manifestation:
IllegalArgumentException (when parsing timeout to an integer)
IOException (when reading the key file)

5. Consequence:
HDFS auto-failover fails, and the entire HDFS service becomes unavailable.

(a) Missing initial checking



Error-handling configuration parameter: Apache httpd-2.4.10

```
CoreDumpDirectory
```

1. LC Errors:
The running program has no permission to access coredump directory.

2. Initial checks: Check if the path points to an **existent directory**.

```
if (apr_stat(&finfo, fname, APR_FINFO_TYPE) != APR_SUCCESS)
  return "CoreDumpDirectory does not exist";
if (finfo.filetype != APR_DIR)
  return "CoreDumpDirectory is not a directory";
```

3. Late execution: Change working directory (**chdir**) to the path.

```
static void sig_coredump(int sig) {
  ...
  apr_filepath_set(ap_coredump_dir, ...);
  ...
}
/* server/mpm_unix.c */

"CoreDumpDirectory"

if(chdir(rootpath) != 0)
  return errno;
```

4. Manifestation:
Error code returned by the chdir call

5. Consequence:
Apache httpd cannot switch to the configured directory, and thus fails to generate the coredump file upon server crashing.

(b) Incomplete initial checking

Figure 3: New LC errors discovered in the latest versions of the studied software, both of which are found to have caused real-world failures [40, 41]. For all these LC errors, the correctness checking is implicitly done when the parameters' values are actually used in operations, which is unfortunately too late to prevent the failures.

adopt the *lazy* practice of using configuration values[3]—parsing and consuming configuration settings only when the values are immediately needed for the operations, without any systematic configuration checking at the system's initialization phase.

With such a practice, even trivial errors could result in big impact on the system's dependability. Figure 3a exemplifies such cases using the new LC errors we discovered in our study. In HDFS, any LC errors (such as a naïve type error) in the auto-failover configurations could break the fail-over procedure upon the NameNode failures (as the values are not checked or used early), making the entire HDFS service become unavailable.

[3]This is a bad but commonly adopted practice in Java and Python programs which rely on libraries (e.g., `java.util.Properties` and `configparser`) to directly retrieve and use configuration values from configuration files on demand, without systematic early checks.

Software	Not used during initialization		Studied param.
HDFS	17	(38.6%)	44
YARN	9	(25.7%)	35
HBase	3	(12.0%)	25
Apache	4	(28.6%)	14
Squid	4	(19.0%)	21
MySQL	6	(13.9%)	43

Table 5: The studied configuration parameters whose values are not used at the system's initialization phase.

Apache, MySQL, and Squid all apply specific configuration checking procedures at initialization, mainly for checking data types and data ranges. However, for more complicated parameters, some checking is incomplete. Figure 3b shows another new LC error we discovered. In this case, though the initial checking code covers file existence and types, it misses other constraints such as file permissions. This leaves Apache subject to permission-related LC errors (which is reported as one common cause of core-dump failures upon server crash [41]).

As shown by Figure 3b, one configuration parameter could have multiple subtle constraints depending on how the system uses its value. For example, a configured file path used by `chdir` has different constraints from files accessed by `open`; even for files accessed by the same `open` call, different flags (e.g., `O_RDONLY` versus `O_CREAT`) would result in different constraints. Implementing code to check such constraints is tedious and error-prone.

Finding 2: *Many (12.0%–38.6%) of the studied RAS configuration parameters are not used at all during the system's initialization phase.*

Table 5 counts the studied configuration parameters that are not used at the system's initialization phase, but are consumed directly in late execution (e.g., when dealing with failures). Figure 3a is such an example. Since all these parameters are from RAS features, it is natural for their usage to come late on demand.

Some Java programs put the checking or usage code of the parameters in the class constructors, so that the errors can be exposed when the class objects are created (specially, this is used as the practice for quickly fixing LC errors [18, 19, 54]). However, this may not fundamentally avoid LC errors if the class objects are not created during the system's initialization phase.

Note: RAS configurations can be implemented with early usage at the system's initialization phase. As shown in Table 5, the majority of RAS configurations are indeed used during initializaiton. For example, all the studied systems choose to `open` error-log files at initialization time, rather than waiting until they have to print the error messages to the log files upon failures.

Software	# RAS Parameters	
	Subject to LC errors	Studied
HDFS	17 (38.6%)	44
YARN	9 (25.7%)	35
HBase	3 (12.0%)	25
Apache	3 (21.4%)	14
Squid	3 (14.3%)	21
MySQL	2 (4.7%)	43
Total	**37 (20.3%)**	**182**

Table 6: The number of configuration parameters that are subject to LC errors in the studied ones. 11 of these parameters have been confirmed/fixed by the developers after we reported them.

Finding 3: *Resulting from Findings 1 and 2, 4.7%–38.6% of the studied RAS parameters do not have any early checks and are thereby subject to LC errors which can cause severe impact on the system's dependability.*

Table 6 shows the number of the RAS configuration parameters that are subject to LC errors in each studied system. The threats are prevalent: LC errors can reside in 10+% of the RAS parameters in five out of six systems. As all these LC errors are discovered in the latest versions, any of them could appear in real deployment and would impair the system's dependability in a latent fashion. Such prevalence of LC errors indicates the need for tool support to systematically rule out the threats.

Among the studied systems, HDFS and YARN have a particularly high percentage of RAS parameters subject to LC errors, due to their lazy evaluation of configuration values (refer to Finding 1 for details). HBase applies the same lazy practice as HDFS and YARN, but has fewer parameters subject to LC errors, because most of its RAS parameters are used during its initialization. We also find LC errors in the other studied systems, despite their initial configuration checking efforts.

2.3 Implication

In summary, even mature software systems are subject to LC errors due to the deficiency of configuration checking at the initialization time. While relying on developers' discipline to add more checking code can help, the reality often fails our expectations, because implementing configuration checking code is tedious and error-prone.

Fortunately, we also observe from the study that except for *explicit* configuration checking code, the actual *usage* of configuration values (which already exists in source code) can serve as an *implicit* form of checking, for example, opening a file path that comes from a configuration value implies a capability check. Such usage-implied checking is often more complete and accurate than the explicit checkers written by developers, because it precisely captures how the configuration values should be used in the actual program execution. Sadly, in reality these usage-implied checking is rarely leveraged to detect LC errors, because the usage often comes too late to be useful. A natural question regarding the solution to LC errors is: can we automatically generate configuration checking code from the existing source code that uses configuration values?

3 PCHECK Design and Implementation

PCHECK is a tool for enabling early detection of configuration errors for a given systems program. The objective of PCHECK is to automatically generate configuration checking code (called *checkers*) based on the original program, and invoke them at the system initialization phase, in order to detect LC errors.

PCHECK tries to generate checkers for every configuration parameter. It is *not* specific to RAS configurations and has *no* assumption on the existence of any LC errors. The checker of a parameter emulates how the system uses the parameter's value in the original execution, and captures anomalies exposed during the emulated execution as the evidence of configuration errors.

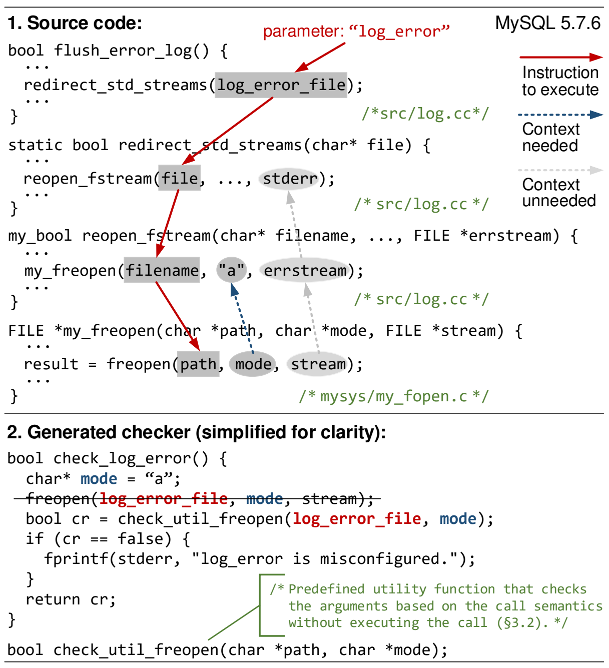
PCHECK is built on top of the Soot [3] and LLVM [4] compiler frameworks and works for both Java and C system programs. PCHECK works on the intermediate representations (IR) of the programs (LLVM IR or Soot Jimple). It takes the original IR as inputs, and outputs the generated checkers, and inserts them into bitcode/bytecode files (which are then built into native binaries). This may require prepending the build process by replacing the compiler front-end with Soot or Clang [47].

PCHECK faces three major challenges: (1) How to automatically emulate the execution that uses configuration values? (2) Since the checkers will be inserted into the original program and will run in the same address space, how does one make the emulation *safe* without incurring side effects on the system's internal state and external environment? (3) How to capture anomalies during the emulated execution as the evidence of configuration errors (the emulation alone cannot directly report errors)?

To address the first challenge, PCHECK extracts the instructions that transform, propagate, and use the value of every configuration parameter using a static taint tracking method. PCHECK then makes a best effort to produce the context (values of dependent variables) necessary for emulating the execution. The extracted instructions, together with the context, are encapsulated in a checker.

For the second challenge, PCHECK "sandboxes" the auto-generated checkers by rewriting instructions that would cause side effects. PCHECK avoids modifications to global variables by copying their values to local ones, and rewrites the instructions that may have external side effects on the underlying OS.

To address the third challenge, PCHECK leverages system- and language-level error identifiers (including runtime exceptions, system-call error codes, and abnor-



```
bool flush_error_log() {
  ...
  redirect_std_streams(log_error_file);
  ...
}                                                /*src/log.cc*/
static bool redirect_std_streams(char* file) {
  ...
  reopen_fstream(file, ..., stderr);
  ...
}                                                /* src/log.cc */
my_bool reopen_fstream(char* filename, ..., FILE *errstream) {
  ...
  my_freopen(filename, "a", errstream);
  ...
}                                                /* src/log.cc */
FILE *my_freopen(char *path, char *mode, FILE *stream) {
  ...
  result = freopen(path, mode, stream);
  ...
}                                                /* mysys/my_fopen.c */
```

```
bool check_log_error() {
  char* mode = “a”;
  ~~freopen(log_error_file, mode, stream);~~
  bool cr = check_util_freopen(log_error_file, mode);
  if (cr == false) {
    fprintf(stderr, "log_error is misconfigured.");
  }
  return cr;
}
                        /* Predefined utility function that checks
                           the arguments based on the call semantics
                           without executing the call (§3.2). */
bool check_util_freopen(char *path, char *mode);
```

Figure 4: Illustration of PCHECK's checker generation (using a real-world LC error example [26]). PCHECK replaces the original call (freopen) with check utilities based on access and stat to prevent side effects (§3.2). To execute the instructions, the necessary execution context needs to be produced. Note that we illustrate the checker using C code for clarity; the actual code is in LLVM IR or Soot Jimple.

mal program exits) to capture the anomalies exposed during the emulation, and report configuration errors.

Figure 4 illustrates PCHECK's checker generation for a MySQL configuration parameter, `log_error`, which is subject to LC errors [26]. PCHECK extracts the instructions that use the configuration value and determines the values of the other dependent variables (e.g., `mode`) as the context. To prevent side effects, it rewrites some call instruction. It detects errors based on the return value.

Lastly, PCHECK inserts the generated checkers into the system program, and invokes these checkers at the end of the system initialization phase (annotated by developers). To detect TOCTTOU errors[4], PCHECK supports running checkers periodically in a separate thread.

Usage. PCHECK requires two inputs from developers: (1) *specifications of the configuration interface* to help PCHECK identify the initial program variables that store configuration values, as the starting points for analysis (§3.1.1); (2) *annotations* of the system's initialization phase where the early checkers will be invoked (§3.4).

In addition, PCHECK provides the tuning interface for developers to select and remove any generated checkers, as per their preference and criteria (e.g., after standard software testing of the enhanced system programs). Similarly, PCHECK provides an operational interface that allows sysadmins to enable/disable the invocation of the checkers of any specific parameters in operation.

3.1 Emulating Execution

To emulate the execution that uses a configuration parameter, PCHECK first identifies instructions that load the parameter's value into program variables (§3.1.1). Starting from there, PCHECK performs forward static taint analysis to extract all the instructions whose execution uses the parameter's value, and hence are the candidates to be included in the checkers (§3.1.2). It then analyzes backwards to figure out the values of dependent variables in these instructions, as the execution context (§3.1.3). Finally, PCHECK composes checkers using the above instructions and their context (§3.1.4).

Note that the emulation does not need to include the conditions under which the configurations are used. Instead, it focuses on executing the instructions that consume the configuration values—the goal is to check if using the configuration would cause any anomalies when its value is needed. With this design, PCHECK is able to effectively handle large, non-deterministic software programs, without the need to inject/simulate hard-to-trigger error conditions under which LC errors are exposed.

3.1.1 Identifying Starting Points

As the configuration-consuming execution always starts from loading the configuration value, PCHECK needs to identify the program variables that initially store the value of each parameter, as the starting points.

PCHECK adopts the common practices presented in previous work [8,22,32,33,52,60,61] to obtain the mapping from configuration parameters to the corresponding variables. The basic idea is to let developers specify the *interface*[5] for retrieving configuration values, and then automatically identify program variables that load the values based on the interface. As pointed out by [52], most mature systems have uniform configuration interfaces for the ease of code maintenance. For instance, to work with HDFS, PCHECK only needs to know the configuration getter functions (e.g., `getInt` and `getString` in Figure 3a) declared in a single Java class; identifying them only requires several lines of specifications using regular expressions. In general, specifying interface requires little specification efforts, compared to annotating every single variable for a large number of configuration parameters. In the evaluation, the specifications needed for most systems are less than 10 lines (§4: Table 7).

[4]A TOCTTOU (Time-Of-Check-To-Time-Of-Use) error occurs after the checking phase and before the use phase, e.g., inadvertently deleting a file that had been checked early but will be used later on.

[5]The interface could be APIs, data structures, or parsing functions [52,35]. It is reported that only three types of interfaces are commonly used to store/retrieve configurations [52,35].

3.1.2 Extracting Instructions Using Configurations

For each configuration parameter, PCHECK extracts the instructions that propagate, transform, and use the parameter's value using a static taint tracking method. For a given parameter, the initial taints are the program variables that store the parameter's value (§3.1.1). The taints are propagated via data-flow dependencies (including assignments, type casts, and arithmetic/string operations), but not through control-flow dependencies to avoid overtainting [39]. All the instructions containing taints are extracted, and will be encapsulated in a checker.

Note that one parameter could be used in multiple execution paths, and thus have multiple checkers. We explain how multiple checkers are aggregated in §3.1.4.

Ordinarily, the extracted instructions from data-flow analysis do not include branches. However, if a tainted instruction is used as a branch condition whose branch body encloses other tainted instructions, PCHECK performs additional control-flow analysis to retain the control dependency of these instructions. One pattern is using a configuration value `p` after a null-pointer check, in the form of, `if (p != NULL) { use p; }`. PCHECK recovers the conditional branch and ensures that if `p`'s value is `NULL`, the instructions using `p` inside the branch would not be reached. Moreover, PCHECK checks if a tainted branch condition leads to abnormal program states, for which it inserts error-reporting instructions (see §3.3).

The taint tracking is inter-procedural, context sensitive, and field sensitive. Inter-procedure is necessary because configuration values are commonly passed through procedure calls, as illustrated in Figure 4. We adopt a summary-based inter-procedural analysis, and assemble the execution based on arguments/returns. PCHECK maintains the call sites; thus it naturally enables context sensitivity which helps produce context by backtracking from callees to callers (c.f., §3.1.3). Field sensitivity is needed as configuration values could be stored in data structures or as class fields. PCHECK scales well for real-world software systems, as configuration-related instructions form a small part of the entire code base. We do not explicitly perform alias analysis (though it is easy to integrate), as configuration variables are seldom aliased.

3.1.3 Producing Execution Context

Some of the extracted instructions that use configuration variables may not be directly executable, if they contain variables that are not defined within the extracted instruction set. To execute such instructions, PCHECK needs to determine the values of these undefined variables (which we refer to as "dependent variables") in order to produce self-contained context.

PCHECK will include a variable and the corresponding instructions in the emulated execution, only when this variable's value stems from configuration values (e.g., `path` in Figure 4) or can be statically determined along the data-flow paths of the configuration value (e.g., `mode` and `stream` in Figure 4). PCHECK does not include dependent variables whose values come from indeterminate global variables, external inputs (from I/O or network operations such as `read` and `recv`), values defined out of the scope of the starting point, etc. For such dependent variables, PCHECK removes the instruction that uses them as operands, together with the succeeding instructions. Those variables' values may not be available during the initialization phase of the system execution; using them would lead to unexpected results.

To produce the context, PCHECK backtracks each undefined dependent variable first intra-procedurally and then inter-procedurally (to handle the arguments of procedure calls). The backtracking starts from the instruction that uses the variable as its operand, and stops until either PCHECK successfully determines the value of the variable or gives up (the value is indeterminate). In Figure 4, PCHECK backtracks `mode` used by the tainted instruction and successfully obtains its value `"a"`.

PCHECK only attempts to produce the minimal context necessary to emulate execution for the purpose of checking. As an optimization, PCHECK is aware of how certain types of instructions will be rewritten in later transformations (e.g., for side-effect prevention, §3.2). In Figure 4's example, PCHECK knows how the `freopen` call will be rewritten later. Therefore, it only produces the context of `mode` which is needed to check the file access; the other dependent variable `stream` is ignored as it is not needed for the checking.

Sometimes, the dependent variables come from other configuration parameters. PCHECK can capture the relationships among multiple configurations, e.g., one parameter's value has to be larger or smaller than another's.

3.1.4 Encapsulation

For each configuration parameter, PCHECK encapsulates the configuration-consuming instructions together with their context into a *checker*, in the form of a *function*. PCHECK clones the original instructions and their operands. For local variables used as operands, PCHECK clones a new local variable and replaces the original variable with the new one. If the instructions change global variables, PCHECK generates a corresponding local variable and copies the global variable's value to the local one (to avoid changing the global program state). When it involves procedure calls, PCHECK inlines the callees.

Handling multiple execution paths. For configuration parameters whose values are used in multiple distinct execution paths, PCHECK generates multiple checkers and aggregates their results. The configuration value is considered erroneous if one of these checkers complains.

PCHECK needs to pay attention to potential path explosion to avoid generating too many checkers. Fortunately, in our experience, configuration values are usually used in a simple and straightforward way, with only a small number of different execution paths to emulate.[6] This makes the PCHECK approach feasible.

Moreover, PCHECK merges two checkers if they are equivalent or if one is equivalent to a subset of the other. PCHECK does this by canonicalizing and comparing the instructions in the checkers' function bodies. Additionally, PCHECK merges checkers which start with the same transformation instruction sequence by reusing the intermediate transformation results.

Note that the checkers with no error identifiers (§3.3) or considered redundant (§3.4) will be abandoned. As shown in §4.4, the number of generated checkers are well bounded, and executing them incurs little overhead.

3.2 Preventing Side Effects

PCHECK ensures that the generated checkers are free of side effects—running the checkers does not change the *internal* program state beyond the checker function itself, or the *external* system environment (e.g., filesystems and OSes). Therefore, PCHECK cannot blindly execute the original instructions. For example, if the checker contains instructions that call `exec`, running the checker would destruct the current process image. Similarly, creating or deleting files is not acceptable, as the filesystem state before and after checking would be inconsistent.

Internal side effects are prevented by design. PCHECK ensures that each checker only has local effects. As discussed in §3.1.4, PCHECK avoids modifying global variables in the checker function; instead, it copies global variable values to local variables and uses the local ones instead. The checker does not manipulate pointers if the pointed values are indeterminate.

External side effects are mainly derived from certain system and library calls that interact with the external environment (e.g., filesystems and OS states). In order to preserve the checking effectiveness without incurring external side effects, PCHECK rewrites the original call instructions to redirect the calls to predefined *check utilities*. A check utility *models* a specific system or library call based on the call semantics. It validates the arguments of the call, but does not actually execute the call. PCHECK implements check utilities for standard APIs and data structures (including system calls, `libc` functions for C, and Java core packages defined in SDK). The check utilities are implemented as libraries that are either statically linked into the system's bitcode (for C programs), or included in the system's `classpath` (for Java programs). In Figure 4, the check utility of `freopen` checks the arguments of the call using `access` and `stat` which are free of side effects (the original `freopen` call will close the file stream specified by the third argument).

PCHECK skips instructions that `read`/`write` file content or `send`/`recv` network packets, in order to stay away from external side effects and heavy checking overhead. Instead, PCHECK performs metadata checks for files and reachability checks for network addresses. This helps the generated checkers be safe and efficient, while still being able to catch a majority of real-world LC errors.

For any library calls that are not defined in PCHECK or do not have known side effects (e.g., some library calls would invoke external programs/commands), PCHECK defensively removes the call instructions (together with the succeeding instructions) to avoid unexpected effects.

One alternative approach to preventing external side effect is to running the checkers inside a sandbox or even a virtual machine at the system initialization phase. This may save the efforts of implementing the check utilies and rewriting system/library call instructions. However, such approach would impair the usability of PCHECK, because it requires additional setups from system administrators in order to run the PCHECK-enhanced program.

3.3 Capturing Anomalies

As the checker emulates the execution that uses the configuration value, anomalies exposed during execution indicate that the value contains errors—the same problem that would occur during real execution. In this case, the checker reports errors and pinpoints the parameter.

PCHECK captures anomalies based on the following three types of *error identifiers*: (1) runtime exceptions that disrupt the emulated execution (for Java programs); (2) error code returned by system and library calls (for C programs); and (3) abnormal program termination and error logging that indicate abnormal program states.

For Java programs, PCHECK captures runtime anomalies based on Java's `Exception` interface, the language's uniform mechanism for capturing error events. PCHECK places the body of the checker function in a `try`/`catch` block. The abnormal execution would throw `Exception` objects and fall into the `catch` block. In this case, the checker reports errors and prints the stack traces.

C programs do not have the uniform error interfaces. Thus, PCHECK leverages the error identifiers defined by specific system/library call semantics, i.e., the return values and `errno`. For example, if the `access` call returns `-1`, it means the call failed when accessing the file (with the reason being encoded in `errno`). In PCHECK, we predefine the error identifiers for commonly-used system and

[6] The emulated execution paths are not the original execution paths (they only include the configuration-related instructions).

libc calls to decide whether a call succeeded or failed. If the call fails, the checker reports configuration errors.

In addition to the anomalies exposed by system and library APIs, a program usually contains hints of abnormal program states. Such hints are instructions such as `exit`, `abort`, `throw`, false assertion, error logging, etc. PCHECK treats these hints as one type of anomalies. If an instruction is post-dominated by any anomaly hints, the instruction itself indicates an abnormal state of execution. Thus, PCHECK reports configuration errors when the checker emulates such error instructions. PCHECK records these hints during the code analysis in §3.1.2, and inserts error-reporting instructions into the checker at the corresponding locations.

PCHECK abandons the checkers that do not contain any of the three types of error identifiers discussed above. In other words, running such checkers cannot expose any explicit anomalies (no evidence of configuration errors).

3.4 Invoking Early Checkers

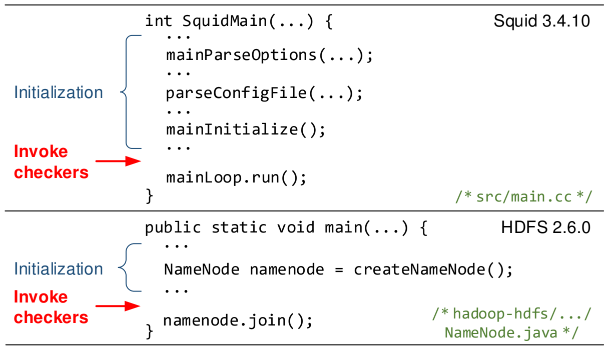
Once the checkers are generated, PCHECK inserts call instructions to invoke the checkers at the program locations specified by developers. The expected location is at the end of the system initialization phase to make the checkers the last defense against LC errors.

Figure 5 shows the locations annotated for PCHECK to invoke the auto-generated checkers for Squid and HDFS. For server systems like Squid, the checkers should be invoked before the server starts to listen and wait for client requests. For distributed systems like HDFS, the checkers should be invoked before the system starts to connect and join the cluster. As all the evaluated systems fall in these two patterns, we believe that specifying the invocation locations is a simple practice for developers.

Some C programs may change user/group identities. Typically, the program starts as `root` and then switches to unprivileged users/groups (e.g., `nobody`) at the end of initialization before handling user requests. In Figure 5, the switch is performed inside `mainInitialize`. As the checkers are invoked in the end of the initialization, the checking results are not affected by user/group switches.

To capture the TOCTTOU errors, PCHECK also supports running the generated checkers periodically in a separate thread. Periodical checking is particularly useful for catching configuration errors that occur after the initial checking (e.g., due to environment changes such as remote host failures and inadvertent file deletion).

Avoiding redundant checking. PCHECK abandons the redundant checkers which are constructed from instructions that would be executed before reaching the invocation location—any configuration errors reported by such checkers should have already been detected by the system's built-in checks, or have been exposed when the configuration value is used, before the checker is called.



```
int SquidMain(...) {
    ...
    mainParseOptions(...);
    ...
    parseConfigFile(...);
    ...
    mainInitialize();
    ...
    mainLoop.run();
}                                   /* src/main.cc */
```

```
public static void main(...) {
    ...
    NameNode namenode = createNameNode();
    ...
    namenode.join();                /* hadoop-hdfs/.../
}                                      NameNode.java */
```

Figure 5: Locations to invoke the checkers in Squid and HDFS NameNode. The auto-generated checkers are expected to be invoked at the end of the initialization phase.

Creating standalone checking programs. Another option to invoking the early checkers is to create a standalone checking program comprised of the checkers, and run it when the configuration file changes. This approach eliminates the need to deal with internal side effect; on the other hand, the checking program is still prohibited to have external side effect. Note that the generated checkers start from the instructions that load configuration values (§3.1.1); therefore, the checking program needs to include the procedures that parse configuration files and store configuration values. This is straightforward for the software systems with modularized parsing procedures[7], but could be difficult if the parsing procedures cannot be easily decoupled from the initialization phase (the initialization may have external side effects).

4 Experimental Evaluation

4.1 Methodology

We first evaluate the effectiveness of PCHECK using the 37 new LC errors discovered in our study. As discussed in §2, all these new LC errors are from the latest versions of the systems; any of them can impair the corresponding RAS features such as fail-over and error handling.

As the design of PCHECK is inspired by the above LC errors, our evaluation contains two more sets of benchmarks to evaluate how PCHECK works beyond these errors. First, we evaluate PCHECK on a distinct set of 21 real-world LC errors that caused system failures in the past. These LC errors are collected from the datasets in prior studies related to configurations [6, 11, 51, 55, 59]; all of them were introduced by real users and caused real-world failures. Some of these cases have different code

[7]We implement this approach for HDFS, YARN, and HBase which use modularized getter functions to parse/store configuration values.

Software	Historical	New	Setup effort
HDFS	7	17	6
YARN	6	9	7
HBase	3	3	6
Apache	2	3	6
Squid	2	3	4
MySQL	1	2	31
Total	21	37	N/A

Table 7: The number of LC error cases used in the evaluation, and the setup efforts (the lines of specifications for identifying starting points, c.f., §3.1.1 and annotations of invocation location, c.f., §3.4).

Type 1:	**Type and format errors (14 cases)**	
Ex. 1:	Ill format settings, e.g., with untrimmed space [14, 16];	
Ex. 2:	Invalid type settings, e.g., 0.05 for integer [13];	
Type 2:	**Undefined options or ranges (6 cases)**	
Ex. 1:	Deprecated compression codec class set by users [15];	
Ex. 2:	Unsupported HTTP protocol settings [17];	
Type 3:	**Incorrect file-path settings (19 cases)**	
Ex. 1:	Non-existent paths which will be opened or executed [37];	
Ex. 2:	Wrong file types, e.g., set regular files for directories [27];	
Type 4:	**Other erroneous settings (19 cases)**	
Ex. 1:	Negative values used by `sleep` and thread `join` [18, 54];	
Ex. 2:	Invalid mail program [38] and unreachable emails [38];	

Table 8: Types and examples of LC errors used in the evaluation.

patterns from the ones we discovered in §2. Table 7 lists the number of these LC errors in each system.

Furthermore, we apply PCHECK to 830 configuration files of the studied systems (except Squid) collected from the official mailing lists of these systems and online technical forums such as ServerFault and StackOverflow [1]. This simulates the experience of using PCHECK on real-world configuration files (§4.2). Moreover, it helps measure the false positive rate of the checking results (§4.6).

Note that we evaluate PCHECK upon all types of LC errors, instead of any specific error types. Therefore, the evaluation results indicate the checking effectiveness of PCHECK in terms of all possible LC errors. Table 8 categorizes and exemplifies the LC errors used in the evaluation based on their types.

Also, the evaluation does not use synthetic errors generated by mutation or fuzzing tools (e.g., ConfErr [23]). Most of the synthetic errors are not LC errors—they are manifested or detected by the system's built-in checks at the system's initialization time. Thus, using such errors would make the results less meaningful to LC errors.

For each system, we apply PCHECK to generate the early checkers and insert them in the system's program. Table 7 lists the setup efforts for the each system evaluated, measured by the lines of specifications for identifying the start points (c.f., §3.1.1) and annotations of the invocation locations (c.f., §3.4). Then, we apply the auto-generated checkers to the configuration files that contain these LC errors. We evaluate the effectiveness of PCHECK based on how many of the real-world LC errors can be reported by the auto-generated checkers.

Types of LC errors	# (%) LC errors detected	
	Historical	New
Type and format error	1/1 (100.0%)	13/13 (100.0%)
Undefined option/range	2/2 (100.0%)	4/4 (100.0%)
Incorrect file/dir path	9/12 (75.0%)	5/7 (71.4%)
Other erroneous setting	3/6 (50.0%)	7/13 (53.8%)
Total	**15/21 (71.4%)**	**29/37 (78.4%)**

Table 9: The number (percentage) of the LC errors detected by the early checkers generated by PCHECK. PCHECK detects 7 (33.3%) and 11 (29.7%) more LC errors among the historical and new LC-error benchmarks respectively, compared to `conf_spellchecker`, a state-of-the-art configuration-error detection tool.

We compare the checking results of PCHECK with `conf_spellchecker` [2, 35], a state-of-the-art static configuration checking tool built on top of automatic type inference of configuration values [33, 35]. For each defined type, `conf_spellchecker` implements corresponding checking functions which are invoked to check the validity of the configuration settings.

4.2 Detecting Real-world LC Errors

PCHECK detects 70+% of both historical and new LC errors (as shown in Table 9), preventing the latent manifestation and resultant system damage imposed by these errors. The results are promising, especially considering that we evaluate PCHECK using all types of configuration errors instead of any specific type. Indeed, PCHECK is by design generic to any types of configuration errors that can be exposed through execution emulation. Many of these LC errors cannot be detected by the state-of-the-art detection tools, as discussed below and in §4.3.

Among the different types of LC errors, PCHECK detects all the errors violating the types/formats and options/ranges constraints. These two types of errors usually go through straightforward code patterns and do not have dependencies with the system's runtime states. For example, most type/format errors in HDFS and YARN are manifested when these systems read and parse the erroneous settings through the getter functions. As the auto-generated checkers invoke the getter instructions, it triggers exceptions and detects the errors.

PCHECK detects the majority of LC errors that violate file-related constraints (including special files such as directories and executables). We observe that the majority of the file parameters fall into recognized APIs, such as `open`, `fopen`, and `FileInputStream`. The undetected file-related LC errors are mainly caused by (1) unknown external usage and (2) indeterminate context. The former prevents the generated checkers from being executed, and the latter stops generation of the checkers. For example, some errors reside in parameters whose values are concatenated into shell command strings, used as the argument of `system()` (to invoke `/bin/sh` to execute the command). As PCHECK has no knowledge of any

Software	# config files	# (%) detected config. errors	
		All	Env. specific
HDFS	245	40	15 (37.5%)
YARN	81	49	32 (65.3%)
HBase	405	139	95 (68.3%)
Apache	65	41	36 (87.8%)
MySQL	34	13	10 (76.9%)

Table 10: Configuration errors detected by applying the checkers on real-world configuration files. Many of the errors can only be detected by considering the system's native environment (§4.3).

Software	# checked param. (# checkers)	All params
HDFS	164 (252)	164
YARN	116 (200)	116
HBase	125 (201)	125
Apache	18 (41)	97
Squid	45 (74)	216
MySQL	32 (51)	462

Table 11: The number of parameters with checkers generated by PCHECK and the total number of generated checkers (each represents a distinct parameter usage scenario).

shell commands, it removes the `system()` call because the side effects are unknown. The other undetected errors are in directories or file prefixes which are merged with dynamic contents from user requests which cannot be obtained statically; thereby, the corresponding checkers cannot be generated. These two causes (unknown external usage and indeterminate context) also account for the undetected errors in the "other" category.

In general, PCHECK is effective in checking errors that are manifested through execution anomalies with error identifiers defined in §3.3, such as those failing at system/library calls or throwing exceptions in the controlled branch. Whereas, it is hard for PCHECK to detect errors defined by application-specific semantics, such as email addresses, internal error code, etc.

We apply `conf_spellchecker` on the same sets of LC errors. Compared with PCHECK, `conf_spellchecker` detects 7 (33.3%) and 11 (29.7%) less LC errors in the historical and new error benchmarks, respectively. The main reason for PCHECK's outperformance is that the execution emulation can achieve fine-grained checking towards high fidelity to the original execution. For example, `conf_spellchecker` can only infer the type of a configuration setting to be a "File". However, it does not understand how the system accesses the file in the execution. Thus, it reports errors if and only if "*the file is neither readable nor writable*" [2]. This heuristic would miss LC errors such as read-only files to be written by the system. Furthermore, type alone only describes a subset of constraints. `conf_spellchecker` misses the LC errors that violate other types of constraints such as data ranges.

4.3 Checking Real-world Configuration Files

We apply the checkers generated by PCHECK to 830 real-world configuration files. PCHECK reports 282 true configuration errors and three false alarms (discussed in §4.6). As shown in Table 10, many (37.5%–87.8%) of the reported configuration errors can only be detected by considering the system's native execution environment. These configuration settings are valid in terms of format and syntax (in fact, they are likely to be correct in the original hosts). However, they are erroneous when used on the current system because the values violate environment constraints such as undefined environment variables, non-existent file paths, unreachable IP addresses, etc. Since PCHECK emulates the execution that uses the configuration values on the system's native execution environment, it naturally detects these errors. On the other hand, such configuration errors are not likely to be detected by traditional detection methods [29,31,36,46,57] that treat configuration values as string literals, and thus are agnostic to the execution environment.

4.4 Checker Generation

Table 11 shows the number of configuration parameters that have checkers generated by PCHECK and the total number of generated checkers for the evaluated systems (multiple checkers could be generated for a parameter).

PCHECK generates checkers for every recognized parameter of HDFS, YARN, and HBase. Each emulated execution in these systems starts from the call instructions of getter functions, so the checkers are able to capture all the errors starting from the parsing phase to the usage phase. For Apache, MySQL and Squid, PCHECK generates fewer checkers. As these systems parse and assign parameter settings to corresponding program variables at the initialization stage, PCHECK bypasses the parsing phase and directly starts from the variables that store the configuration value. Since a large number of the Boolean and numeric variables are only used for branch control with no error identifier (both branches are valid), PCHECK does not generate checkers for them (c.f., §3.3). Moreover, many of the variables are only used at the initialization phase before reaching the invocation location, so their checkers are considered redundant and thus are abandoned (c.f., §3.4).

The other issues that prevent checker generation include dependencies on the system's runtime states and uses of customized APIs (e.g., Apache uses customized `APR` string operations which heavily rely on predefined memory pools). Fortunately, as shown in §4.2, the majority of the LC errors have standard code patterns and can be detected using PCHECK's approach. Generating checkers for the rest of the errors require more advanced analysis and program-specific semantics.

Also, we can see that the total number of checkers are well bounded, which is attributable to the execution merging (§3.1.4) and redundancy elimination (§3.4).

Software	Time for running the checkers (millisec.)			
HDFS	[NameNode]	408	[DataNode]	311
YARN	[ResourceMgr]	243	[NodeMgr]	486
HBase	[HMaster]	780	[RegionServer]	777
Apache	[httpd]	0.6	————	——
Squid	[squid]	93.8	————	——
MySQL	[mysqld]	1.7	————	——

Table 12: Checking overhead (measured by the time needed to run the auto-generated checkers).

4.5 Checking Overhead

The checkers are only invoked at the initialization phase or run in a separate thread, thus they have little impact on the systems' runtime performance. We measure their overhead to be the time needed to execute these checkers, by inserting time counters before and after invoking all the checkers. Table 12 shows the time in milliseconds (ms) to run the checkers on a 4-core, 2.50GHz processor connected to a local network (for distributed systems like HDFS, YARN, and HBase, the peer nodes are located in the same local network). The checking overhead for Apache and MySQL is negligible (less than 5ms); Squid needs around 100ms because it has a parameter that points to public IP addresses (`announce_host`). The overhead for the three Java programs is less than a second. The main portion of the time is spent on network- and file-related checking. Since PCHECK only performs lightweight checks (e.g., metadata checks and reachability checks), the overhead is small. Note that the checkers are currently executed sequentially. It is straightforward to invoke multiple checkers in parallel to reduce overhead, as all the checkers are independent.

4.6 False Positives

We measure false positives by applying the checkers generated by PCHECK to both the default configuration values of the evaluated systems and the 830 real-world configuration files, and examine whether or not our checkers would falsely report errors. We also manually inspect the code of the generated checkers in LLVM IR and Jimple to look for potential incorrectness.

Among all the configuration parameters in the evaluated systems, only three of them have false alarms reported by the auto-generated checkers: two from YARN and one from HBase. All these false positives are caused by the checkers incorrectly skipping conditional instructions affected by the configuration value (§3.1.2), due to unsound static analysis that misses control dependencies. This results in emulating the execution that should never happen in reality—certainly, the anomalies exposed in such execution are unreal. The overall false positive rates are low. YARN has the most configuration parameters with false checkers, with the false positive rate of 1.7% (2 over 116 parameters). Note that checkers with false positives can be removed by the developers or disabled by the administrators in the field (c.f., §3: Usage).

5 Limitations

No tool is perfect. PCHECK is no exception. Like many other error detection tools, PCHECK is neither sound nor complete for its checking scope and the design trade-offs.

PCHECK targets on the specific type of configuration errors which are manifested through explicit, recognizable instruction-level anomalies (c.f., §3.3). It cannot detect *legal* misconfigurations [55] that have valid values but do not deliver the intended system behavior. The common legal misconfigurations include inappropriate configuration settings that violate resource constraints or performance requirements (e.g., insufficient heap size and too small timeout). Such misconfigurations are notoriously hard to detect and are often manifested in a latent fashion as well, such as runtime out-of-memory errors [10] (resources are not used up immediately). However, detecting resource- and performance-related misconfigurations would need dynamic information regarding resource usage and performance profiling, which is beyond the static methods of PCHECK.

In addition, PCHECK cannot emulate the execution that depends on runtime inputs/workloads, or does not have statically determinate context in the program code (c.f., §3.1.3). Thus, it would miss the configuration errors that are only manifested during such execution. Nevertheless, indeterminate context (e.g., those derived from inputs and workloads) can potentially be modeled with representative values, which could significantly improve the capability of checker generation.

One design choice we make is to trade soundness for safety and efficiency—PCHECK aims to detect common LC errors without incurring side effects or much overhead. For example, PCHECK does not look into file contents but only checks if the file can be accessed as expected. Similarly, PCHECK only checks the reachability of a configured IP address or host instead of connecting and sending packets to the remote host. It is possible that certain sophisticated errors can escape from PCHECK (e.g., the configured file is corrupted and thus has wrong contents). As the first step, we target on basic, common errors, as they already account for a large number of real-world LC errors [24, 43, 55]. Efficiently detecting sophisticated errors may require not only deeper analysis but also application semantics.

6 Concluding Remarks

This paper advocates early detection of configuration errors to minimize failure damage, especially in cloud and data-center systems. Despite all the efforts of validation, review, and testing, configuration errors (even those obvious errors) still cause many high-impact incidents of today's Internet and cloud systems. We believe that this is partly due to the lack of automatic solutions for cloud and data-center systems to detect and defend against configuration errors (the existing solutions are hard to be applied, due to their strong reliance on datasets).

We envisage that PCHECK is the first step towards a generic and systematic solution to detect configuration errors. PCHECK does not require collecting any external datasets and is not specific to any specific rules. It detects configuration errors based on how the system actually uses the configuration values. With PCHECK, we demonstrate that such detection method can effectively detect the majority (75+%) of real-world LC errors, with little runtime overhead and setup effort.

7 Acknowledgement

We greatly appreciate the anonymous reviewers and our shepherd, Peter M. Chen, for their insightful comments and feedback. We thank the Opera group, the UCSD Systems and Networking group, and Shelby Thomas for useful discussions and paper proofreading. Tao Cai participated in the implementation of PCHECK. Liqiong Yang contributed to the study of RAS related configuration parameters. Yuanyuan Zhou's group is supported in part by NSF grants (CCR-1526966, CCR-1321006), and a gift grant from Facebook, and supports from NetApp. Shan Lu's research is supported in part by NSF grants (IIS-1546543, CNS-1563956, CNS-1514256, CCF-1514189, CCF-1439091), and generous supports from Alfred P. Sloan Foundation and Google Faculty Research Award.

References

[1] Configuration Datasets. https://github.com/tianyin/configuration_datasets.

[2] conf_spellchecker. https://github.com/roterdam/jchord/tree/master/conf_spellchecker.

[3] Soot: a Java Optimization Framework. http://sable.github.io/soot/.

[4] The LLVM Compiler Infrastructure Project. http://llvm.org/.

[5] ATTARIYAN, M., CHOW, M., AND FLINN, J. X-ray: Automating Root-Cause Diagnosis of Performance Anomalies in Production Software. In *Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation (OSDI'12)* (Hollywood, CA, USA, Oct. 2012).

[6] ATTARIYAN, M., AND FLINN, J. Automating Configuration Troubleshooting with Dynamic Information Flow Analysis. In *Proceedings of the 9th USENIX Conference on Operating Systems Design and Implementation (OSDI'10)* (Vancouver, BC, Canada, Oct. 2010).

[7] BARROSO, L. A., AND HÖLZLE, U. *The Datacenter as a Computer: An Introduction to the Design of Warehouse-scale Machines*. Morgan and Claypool Publishers, 2009.

[8] BEHRANG, F., COHEN, M. B., AND ORSO, A. Users Beware: Preference Inconsistencies Ahead. In *Proceedings of the 10th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC/FSE'15)* (Bergamo, Italy, Aug. 2015).

[9] BRODKIN, J. Why Gmail went down: Google misconfigured load balancing servers. http://arstechnica.com/information-technology/2012/12/why-gmail-went-down-google-misconfigured-chromes-sync-server/.

[10] FANG, L., NGUYEN, K., XU, G., DEMSKY, B., AND LU, S. Interruptible Tasks: Treating Memory Pressure As Interrupts for Highly Scalable Data-Parallel Programs. In *Proceedings of the 25th Symposium on Operating System Principles (SOSP'15)* (Monterey, CA, USA, Oct. 2015).

[11] GUNAWI, H. S., HAO, M., LEESATAPORNWONGSA, T., PATANA-ANAKE, T., DO, T., ADITYATAMA, J., ELIAZAR, K. J., LAKSONO, A., LUKMAN, J. F., MARTIN, V., AND SATRIA, A. D. What Bugs Live in the Cloud? A Study of 3000+ Issues in Cloud Systems. In *Proceedings of the 5th ACM Symposium on Cloud Computing (SoCC'14)* (Seattle, WA, USA, Nov. 2014).

[12] HADOOP ISSUE #134. JobTracker trapped in a loop if it fails to localize a task. https://issues.apache.org/jira/browse/HADOOP-134.

[13] HADOOP ISSUE #2081. Configuration getInt, getLong, and getFloat replace invalid numbers with the default value. https://issues.apache.org/jira/browse/HADOOP-2081.

[14] HADOOP ISSUE #6578. Configuration should trim whitespace around a lot of value types. https://issues.apache.org/jira/browse/HADOOP-6578.

[15] HADOOP-USER MAILING LIST ARCHIVES. Compression codec com.hadoop.compression.lzo.LzoCodec not found. http://goo.gl/N9XFvt.

[16] HBASE ISSUE #6973. Trim trailing whitespace from configuration values. https://issues.apache.org/jira/browse/HBASE-6973.

[17] HDFS ISSUE 5872#. Validate configuration of dfs.http.policy. https://issues.apache.org/jira/browse/HDFS-5872.

[18] HDFS ISSUE #7726. Parse and check the configuration settings of edit log to prevent runtime errors. https://issues.apache.org/jira/browse/HDFS-7726.

[19] HDFS ISSUE #7727. Check and verify the auto-fence settings to prevent failures of auto-failover. https://issues.apache.org/jira/browse/HDFS-7727.

[20] HUANG, P. *Understanding and Dealing with Failures in Cloud-Scale Systems*. PhD thesis, University of California San Diego, Computer Science and Engineering, 2016.

[21] Jiang, W., Hu, C., Pasupathy, S., Kanevsky, A., Li, Z., and Zhou, Y. Understanding Customer Problem Troubleshooting from Storage System Logs. In *Proceedings of the 7th USENIX Conference on File and Storage Technologies (FAST'09)* (San Francisco, CA, USA, Feb. 2009).

[22] Jin, D., Cohen, M. B., Qu, X., and Robinson, B. PrefFinder: Getting the Right Preference in Configurable Software Systems. In *Proceedings of the 29th IEEE/ACM International Conference on Automated Software Engineering (ASE'14)* (Västerås, Sweden, Sep. 2014).

[23] Keller, L., Upadhyaya, P., and Candea, G. ConfErr: A Tool for Assessing Resilience to Human Configuration Errors. In *Proceedings of the 38th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN'08)* (Anchorage, AK, USA, Jun. 2008).

[24] Maurer, B. Fail at Scale: Reliability in the Face of Rapid Change. *Communications of the ACM 58*, 11 (Nov. 2015), 44–49.

[25] Moskowitz, A. Software Testing for Sysadmin Programs. *USENIX ;login: 40*, 2 (Apr. 2015), 37–45.

[26] MySQL Bug #74720. No warn/error message if "log-error" is misconfigured (causing latent log loss). http://bugs.mysql.com/bug.php?id=74720.

[27] MySQL Bug #75645. Runtime Error Caused by Misconfigured BackupDataDir. http://bugs.mysql.com/bug.php?id=75645.

[28] Oppenheimer, D., Ganapathi, A., and Patterson, D. A. Why Do Internet Services Fail, and What Can Be Done About It? In *Proceedings of the 4th USENIX Symposium on Internet Technologies and Systems (USITS'03)* (Seattle, WA, USA, Mar. 2003).

[29] Palatin, N., Leizarowitz, A., Schuster, A., and Wolff, R. Mining for Misconfigured Machines in Grid Systems. In *Proceedings of the 12th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD'06)* (Philadelphia, PA, USA, Aug. 2006).

[30] Patterson, D., Brown, A., Broadwell, P., Candea, G., Chen, M., Cutler, J., Enriquez, P., Fox, A., Kiciman, E., Merzbacher, M., Oppenheimer, D., Sastry, N., Tetzlaff, W., Traupman, J., and Treuhaft, N. Recovery-Oriented Computing (ROC): Motivation, Definition, Techniques, and Case Studies. Tech. Rep. UCB//CSD-02-1175, University of California Berkeley, Mar. 2002.

[31] Potharaju, R., Chan, J., Hu, L., Nita-Rotaru, C., Wang, M., Zhang, L., and Jain, N. ConfSeer: Leveraging Customer Support Knowledge Bases for Automated Misconfiguration Detection. In *Proceedings of the 35th International Conference on Very Large Data Bases (VLDB'15)* (Kohala Coast, HI, USA, Aug. 2015).

[32] Rabkin, A., and Katz, R. Precomputing Possible Configuration Error Diagnosis. In *Proceedings of the 26th IEEE/ACM International Conference on Automated Software Engineering (ASE'11)* (Lawrence, KS, USA, Nov. 2011).

[33] Rabkin, A., and Katz, R. Static Extraction of Program Configuration Options. In *Proceedings of the 33th International Conference on Software Engineering (ICSE'11)* (Honolulu, HI, USA, May 2011).

[34] Rabkin, A., and Katz, R. How Hadoop Clusters Break. *IEEE Software Magazine 30*, 4 (Jul. 2013), 88–94.

[35] Rabkin, A. S. *Using Program Analysis to Reduce Misconfiguration in Open Source Systems Software*. PhD thesis, University of California, Berkeley, 2012.

[36] Santolucito, M., Zhai, E., and Piskac, R. Probabilistic Automated Language Learning for Configuration Files. In *28th International Conference on Computer Aided Verification (CAV'16)* (Toronto, Canada, Jul. 2016).

[37] Squid Bug #1703. diskd related 100% CPU after 'squid -k rotate'. http://bugs.squid-cache.org/show_bug.cgi?id=1703.

[38] Squid Bug #4186. The mail notification feature is buggy and does not deal with configuration errors. http://bugs.squid-cache.org/show_bug.cgi?id=4186.

[39] Sridharan, M., Fink, S. J., and Bodík, R. Thin Slicing. In *Proceedings of the 28th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'07)* (San Diego, CA, USA, Jun. 2007).

[40] StackOverflow Question #21253299. Hadoop sshfence (permission denied). http://stackoverflow.com/questions/21253299/hadoop-sshfence-permission-denied.

[41] StackOverflow Question #7732983. Core dump file is not generated. http://stackoverflow.com/questions/7732983/core-dump-file-is-not-generated.

[42] Sverdlik, Y. Microsoft: Misconfigured Network Device Led to Azure Outage. http://www.datacenterdynamics.com/focus/archive/2012/07/microsoft-misconfigured-network-device-led-azure-outage, 2012.

[43] Tang, C., Kooburat, T., Venkatachalam, P., Chander, A., Wen, Z., Narayanan, A., Dowell, P., and Karl, R. Holistic Configuration Management at Facebook. In *Proceedings of the 25th Symposium on Operating System Principles (SOSP'15)* (Monterey, CA, USA, Oct. 2015).

[44] The Availability Digest. Poor Documentation Snags Google. http://www.availabilitydigest.com/public_articles/0504/google_power_out.pdf.

[45] Thomas, K. Thanks, Amazon: The Cloud Crash Reveals Your Importance. http://www.pcworld.com/article/226033/thanks_amazon_for_making_possible_much_of_the_internet.html.

[46] Wang, H. J., Platt, J. C., Chen, Y., Zhang, R., and Wang, Y.-M. Automatic Misconfiguration Troubleshooting with PeerPressure. In *Proceedings of the 6th*

USENIX Conference on Operating Systems Design and Implementation (OSDI'04) (San Francisco, CA, USA, Dec. 2004).

[47] WANG, X., ZELDOVICH, N., KAASHOEK, M. F., AND SOLAR-LEZAMA, A. Towards optimization-safe systems: Analyzing the impact of undefined behavior. In *Proceedings of the 24th Symposium on Operating System Principles (SOSP'13)* (Farmington, PA, USA, Nov. 2013).

[48] WANG, Y.-M., VERBOWSKI, C., DUNAGAN, J., CHEN, Y., WANG, H. J., YUAN, C., AND ZHANG, Z. STRIDER: A Black-box, State-based Approach to Change and Configuration Management and Support. In *Proceedings of the 17th Large Installation Systems Administration Conference (LISA'03)* (San Diego, CA, USA, Oct. 2003).

[49] WHITAKER, A., COX, R. S., AND GRIBBLE, S. D. Configuration Debugging as Search: Finding the Needle in the Haystack. In *Proceedings of the 6th USENIX Conference on Operating Systems Design and Implementation (OSDI'04)* (San Francisco, CA, USA, Dec. 2004).

[50] WIKIPEDIA. Reliability, availability and serviceability (computing). https://en.wikipedia.org/wiki/Reliability,_availability_and_serviceability_(computing), 2010.

[51] XU, T., JIN, L., FAN, X., ZHOU, Y., PASUPATHY, S., AND TALWADKER, R. Hey, You Have Given Me Too Many Knobs! Understanding and Dealing with Over-Designed Configuration in System Software. In *Proceedings of the 10th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC/FSE'15)* (Bergamo, Italy, Aug. 2015).

[52] XU, T., ZHANG, J., HUANG, P., ZHENG, J., SHENG, T., YUAN, D., ZHOU, Y., AND PASUPATHY, S. Do Not Blame Users for Misconfigurations. In *Proceedings of the 24th Symposium on Operating System Principles (SOSP'13)* (Farmington, PA, USA, Nov. 2013).

[53] XU, T., AND ZHOU, Y. Systems Approaches to Tackling Configuration Errors: A Survey. *ACM Computing Surveys (CSUR) 47*, 4 (Jul. 2015).

[54] YARN ISSUE #2166. Timelineserver should validate that yarn.timeline-service.leveldb-timeline-store.ttl-interval-ms is greater than zero when level db is for timeline store. https://issues.apache.org/jira/browse/YARN-2166.

[55] YIN, Z., MA, X., ZHENG, J., ZHOU, Y., BAIRAVASUNDARAM, L. N., AND PASUPATHY, S. An Empirical Study on Configuration Errors in Commercial and Open Source Systems. In *Proceedings of the 23rd ACM Symposium on Operating Systems Principles (SOSP'11)* (Cascais, Portugal, Oct. 2011).

[56] YUAN, C., LAO, N., WEN, J.-R., LI, J., ZHANG, Z., WANG, Y.-M., AND MA, W.-Y. Automated Known Problem Diagnosis with Event Traces. In *Proceedings of the 1st EuroSys Conference (EuroSys'06)* (Leuven, Belgium, Apr. 2006).

[57] YUAN, D., XIE, Y., PANIGRAHY, R., YANG, J., VERBOWSKI, C., AND KUMAR, A. Context-based Online Configuration Error Detection. In *Proceedings of 2011 USENIX Annual Technical Conference (USENIX ATC'11)* (Portland, OR, USA, Jun. 2011).

[58] ZHAI, E., CHEN, R., WOLINSKY, D. I., AND FORD, B. Heading Off Correlated Failures through Independence-as-a-Service. In *Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation (OSDI'14)* (Broomfield, CO, USA, Oct. 2014).

[59] ZHANG, J., RENGANARAYANA, L., ZHANG, X., GE, N., BALA, V., XU, T., AND ZHOU, Y. EnCore: Exploiting System Environment and Correlation Information for Misconfiguration Detection. In *Proceedings of the 19th International Conference on Architecture Support for Programming Languages and Operating Systems (ASPLOS'14)* (Salt Lake City, UT, USA, Mar. 2014).

[60] ZHANG, S., AND ERNST, M. D. Automated Diagnosis of Software Conguration Errors. In *Proceedings of the 35th International Conference on Software Engineering (ICSE'13)* (San Francisco, CA, USA, May 2013).

[61] ZHANG, S., AND ERNST, M. D. Which Configuration Option Should I Change? In *Proceedings of the 36th International Conference on Software Engineering (ICSE'14)* (Hyderabad, India, May 2014).

[62] ZHANG, S., AND ERNST, M. D. Proactive Detection of Inadequate Diagnostic Messages for Software Configuration Errors. In *Proceedings of the 2015 International Symposium on Software Testing and Analysis (ISSTA'15)* (Baltimore, MD, USA, Jul. 2015).

Kraken: Leveraging Live Traffic Tests to Identify and Resolve Resource Utilization Bottlenecks in Large Scale Web Services

Kaushik Veeraraghavan Justin Meza David Chou Wonho Kim Sonia Margulis
Scott Michelson Rajesh Nishtala Daniel Obenshain Dmitri Perelman
Yee Jiun Song

Facebook Inc.

Abstract

Modern web services such as Facebook are made up of hundreds of systems running in geographically-distributed data centers. Each system needs to be allocated capacity, configured, and tuned to use data center resources efficiently. Keeping a model of capacity allocation current is challenging given that user behavior and software components evolve constantly.

Three insights motivate our work: (1) the live user traffic accessing a web service provides the most current target workload possible, (2) we can empirically test the system to identify its scalability limits, and (3) the user impact and operational overhead of empirical testing can be largely eliminated by building automation which adjusts live traffic based on feedback.

We build on these insights in Kraken, a new system that runs load tests by continually shifting live user traffic to one or more data centers. Kraken enables empirical testing by monitoring user experience (e.g., latency) and system health (e.g., error rate) in a feedback loop between traffic shifts. We analyze the behavior of individual systems and groups of systems to identify resource utilization bottlenecks such as capacity, load balancing, software regressions, performance tuning, and so on, which can be iteratively fixed and verified in subsequent load tests. Kraken, which manages the traffic generated by 1.7 billion users, has been in production at Facebook for three years and has allowed us to improve our hardware utilization by over 20%.

1 Introduction

Modern web services comprise software systems running in multiple data centers that cater to a global user base. At this scale, it is important to use all available data center resources as efficiently as possible. Effective resource utilization is challenging because:

- **Evolving workload**: The workload of a web service is constantly changing as its user base grows and new products are launched. Further, individual software systems might be updated several times a day [35] or even continually [27]. While modeling tools [20, 24, 46, 51] can estimate the initial capacity needs of a system, an evolving workload can quickly render models obsolete.
- **Infrastructure heterogeneity**: Constructing data centers at different points in time leads to a variety of networking topologies, different generations of hardware, and other physical constraints in each location that each affect how systems scale.
- **Changing bottlenecks**: Each data center runs hundreds of software systems with complex interactions that exhibit *resource utilization bottlenecks*, including performance regressions, load imbalance, and resource exhaustion, at a variety of scales from single servers to entire data centers. The sheer size of the system makes it impossible to understand all the components. In addition, these systems change over time, leading to different bottlenecks presenting themselves. Thus, we need a way to continually identify and fix bottlenecks to ensure that the systems scale efficiently.

Our key insight is that the live user traffic accessing a web service provides the most current workload possible, with natural phenomena like non-uniform request arrival rates. As a baseline, the web service must be capable of executing its workload within a preset latency and error threshold. Ideally, the web service should be capable of handling peak load (e.g., during Super Bowl) when unexpected bottlenecks arise, and still achieve good performance.

We propose Kraken, a new system that allows us to run live traffic load tests to accurately assess the capacity of a complex system. In building Kraken, we found that reliably tracking system health is the most important requirement for live traffic testing. But how do we select among thousands of candidate metrics? We aim to provide a good experience to all users by ensuring that a user served out of a data center undergoing a Kraken test has a comparable experience to users being served out of any other data center. We accomplish this goal with a light-weight and configurable monitoring component seeded with two topline metrics, the web server's 99th percentile response time and HTTP fatal error rate, as reliable proxies for user experience.

We leverage Kraken as part of an iterative methodology to improve capacity utilization. We begin by running a load test that directs user traffic at a target cluster or region. A successful test concludes by hitting the utilization targets without crossing the latency or error rate thresholds. Tests can fail in two ways:

- We fail to hit the target utilization or exceed a preset threshold. This is the usual outcome following which we drill into test data to identify bottlenecks.
- Rarely, the load test results in a HTTP error spike or some similar unexpected failure. When this occurs, we analyze the test data to understand what monitoring or system understanding we were missing. In practice, sudden jumps in HTTP errors or other topline metrics almost never happen; the set of auxiliary metrics we add to our monitoring are few in number and are the result of small incidents rather than catastrophic failure.

Each test provides a probe into an initially uncharacterized system, allowing us to learn new things. An unsuccessful test provides data that allows us to either make the next test safer to run or increase capacity by removing a bottleneck. As tests are non-disruptive to users, we can run them regularly to determine both a data center's maximal load (e.g., requests per second, which we term the system's *capacity*), and continually identify and fix bottlenecks to improve system utilization.

Our first year of operating Kraken in production exposed several challenges. The first was that the systems often exhibited a non-linear response where a small traffic shift directed at a data center could trigger an error spike. Our follow-up was to check the health of all the major systems involved in serving user traffic to determine which ones were affected most during the test. This dovetailed into our second challenge which was that systems have complex dependencies so it was often unclear which system initially failed and then trigged errors in seemingly unrelated downstream subsystems.

We addressed both of these challenges by encouraging subsystem developers to identify system-specific counters for performance (e.g., response quality), error rate, and latency, that could be monitored during a test. We focused on these three metrics because they represent the contracts that clients of a service rely on—we have found that nearly every production system wishes to maintain or decrease its latency and error rate while maintaining or improving performance.

The third challenge was one of culture. Although we ensure the tests are safe for users, load tests put significant stress on many systems. Rather than treat load tests as painful events for systems to survive, we worked hard to create a collaborative environment where developers looked forward to load tests to better understand their systems. We planned tests ahead of time and communicated schedules widely to minimize surprise. We encouraged engineers to share tips on how to better monitor their systems, handle different failure scenarios, and build out levers to mitigate production issues. Further, we used a shared IRC channel to track tests and debug issues live. Often, we would use tests as an opportunity for developers to test or validate improvements, which made them an active part of the testing process. Engaging engineers was critical for success; engaged engineers improve their systems quickly and support a frequent test cadence, allowing us to iterate quickly.

Our initial remediations to resolve bottlenecks were mostly capacity allocations to the failing system. As our understanding of Facebook's systems has improved, our mitigation toolset has also expanded to include configuration and performance tuning, load balancing improvements, profiling-guided software changes and occasionally a system redesign. Our monitoring has also improved and as of October 2016, we have identified metrics from 23 critical systems that are bellwethers of non-linear behavior in our infrastructure. We use these metrics as inputs to control Kraken's behavior.

Kraken has been in use at Facebook for over three years and has run thousands of load tests on production systems using live user traffic. Our contributions:

- Kraken is the first live traffic load testing framework to test systems ranging in size from individual servers to entire data centers, to our knowledge.
- Kraken has allowed us to iteratively increase the capacity of Facebook's infrastructure with an empirical approach that improves the utilization of systems at every level of the stack.
- Kraken has allowed us to identify and remediate regressions, address load imbalance and resource exhaustion across Facebook's fleet. Our initial tests stopped at about 70% of theoretical capacity, but now routinely exceed 90%, providing a 20% increase in request serving capacity.

The Kraken methodology is not applicable to all services. These assumptions/caveats underpin our work:

- *Assumption: stateless servers.* We assume that *stateless* servers handle requests without using *sticky sessions* for server affinity. Stateless web servers provide high availability despite system or network failures by routing requests to any available machine. Note that stateless servers may still communicate with services that *are* stateful (such as a database) when handling a request.
- *Caveat: load must be controllable by re-routing requests.* Subsystems that are global in nature may be insensitive to where load enters the system. Such systems may include batch processors, message queues, storage, etc. Kraken is not designed as a comprehensive capacity assessment tool for every

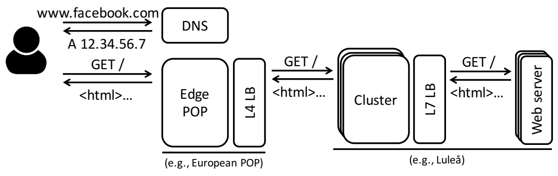


Figure 1: When a user sends a request to Facebook, a DNS resolver points the request at an edge point-of-presence (POP) close to the user. A L4 and L7 load balancer forward the request to a particular data center and a web server respectively.

subsystem. These systems often need more specialized handling to test their limits.

- *Assumption: downstream services respond to shifts in upstream service load.* Consider the case of a web server querying a database. A database with saturated disk bandwidth affects the number of requests served by the web server. This observation extends to other system architectures such as aggregator–leaf where a stateless aggregation server collects data from a pool of leaf servers to service a request.

2 The case for live traffic load tests

Broadly, two common approaches exist for identifying resource utilization bottlenecks: (1) *load modeling (or simulation)* and (2) *load testing (or benchmarking)*.

Load modeling relies on analytical models of data center resources to examine the trade-offs between performance, reliability, and energy-efficiency [11, 13, 18, 40]. We argue that it is infeasible to accurately model a large scale web service's capacity given their evolving workload, frequent software release cycles, and complexity of dependencies [27, 35], Alternatively, load test suites such as TPC-C [44], YCSB [49] and JouleSort [34] use system-level benchmarks to measure the load a system can sustain. Unfortunately, their synthetic workloads only cover very specific use cases, e.g., TPC-C performs a certain set of queries on a SQL database with a fixed set of inputs. An alternate design choice is to use *shadow* traffic where an incoming request is logged and replayed in a test environment. For the web server use case, most operations have side-effects that propagate deep into the system. Shadow tests must not trigger these side effects, as doing so can alter user state. Stubbing out side effects for shadow testing is not only impractical due to frequent changes in server logic, but also reduces the fidelity of the test by not stressing dependencies that would have otherwise been affected.

In contrast, Kraken uses *live* traffic to perform load tests. We prefer live traffic tests because:

- Live user traffic is a fully representative workload that consists of both read and write requests with a non-uniform arrival pattern, and traffic bursts.
- Live traffic tests can be run on production systems without requiring alternate test setups. Further, live traffic tests can expose bottlenecks that arise due to complex system dependencies, which are hard to reproduce in small scale test setups.
- Live traffic load tests on production systems have the implicit benefit of forcing teams to harden their systems to handle traffic bursts, overloads, etc., thus increasing the system's resilience to faults.

Safety is a key constraint when working with live traffic on a production system. We identify two problematic situations: (1) *internal* faults in system operation, such as violation of performance, reliability, or other constraints; and (2) *external* faults that result in capacity reduction due to, for example, network partitions, or power loss. Both situations require careful monitoring and fast traffic adjustment to safeguard the production system.

3 Design

Kraken is constructed as a feedback loop that shifts user traffic to evaluate the capacity of the system under test and identify resource utilization bottlenecks.

3.1 Traffic shifting

Figure 1 provides an overview of how a user request to Facebook is served. The user's request is sent to their ISP, which contacts a DNS resolver to map the URL to an IP address. This IP address maps to one of tens of edge point-of-presence (POP) locations distributed worldwide. A POP consists of a small number of servers on the edge of the network typically co-located with a local internet service provider. The user's SSL session is terminated in a POP at a L7 load balancer, which then forwards the request to one of the data centers.

At Facebook, we group 1–3 data centers in close proximity into a "region". Within each data center, we group machines into one or more logical "frontend" clusters of web servers, "backend" clusters of storage systems, and

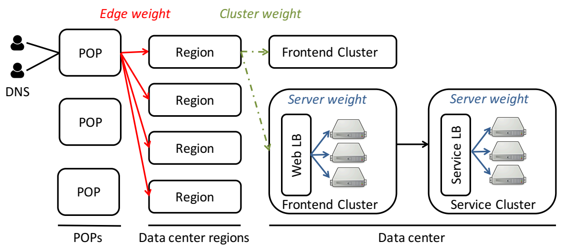


Figure 2: This figure provides an overview of the traffic management components at Facebook. User requests arrive at Edge POPs (points-of-presence). A series of weights defined at the edge, cluster, and server levels are used to route user requests from a POP to a web server in one of Facebook's data centers.

multiple "service" clusters. In this paper, we define a "service" as a set of sub-systems that provide a particular product either internally within Facebook's infrastructure or externally to end users. Each cluster has a few thousand generally heterogeneous machines. Many services span clusters, but web server deployments are confined to a single cluster.

As Figure 2 shows, the particular frontend cluster that a request is routed to depends on two factors: (1) the *edge weight* from a POP to a region, and (2) the *cluster weight* assigned to each frontend cluster in a region. To understand why we need edge weights, consider a request from a user in Hamburg that is terminated at a hypothetical POP in Europe. This POP might prefer forwarding user requests to the Luleå, Sweden region rather than Forest City, North Carolina to minimize latency, implying that the Europen POP could assign a higher edge weight to Luleå than Forest City. A data center might house multiple frontend clusters with machines from different hardware generations. The capability of a Haswell cluster will exceed that of a Sandybridge cluster, resulting in differing cluster weights as well as individual servers being assigned different *server weights*.

When a request reaches a frontend cluster, an L7 load balancer forwards the request to one of many thousands of web servers. The web server may communicate with tens or hundreds of services residing in one or more service clusters to gather the data required to generate a response. Finally, the web server sends the response back to the POP, which forwards the response to the user.

Because web servers and some services in this architecture are stateless, any such server can handle any request bound for it or its peers in the same service. This implies that edge weights, cluster weights, and server weights can be programmed to easily and quickly change their destinations for requests. This allows us to programmatically shift different amounts of load to a particular region or cluster. We use live traffic shifting as the mechanism to manipulate load on different resources. Shifting live traffic allows us to gauge aggregate system capacity in a realistic environment.

3.2 Monitoring

The most important requirements in live traffic testing are reliable metrics that track the health of the system. At Facebook, we use Gorilla [33], a time series database that provides a fast way to store and query aggregated metrics. Our intuition when developing Kraken was that it could query Gorilla for the metrics of all important systems and use the results to compute the next traffic shift.

Our intuition proved impractical as Facebook is composed of thousands of systems, each of which export dozens of counters tracking their input request rate, internal state, and output. One of the goals of Kraken is to

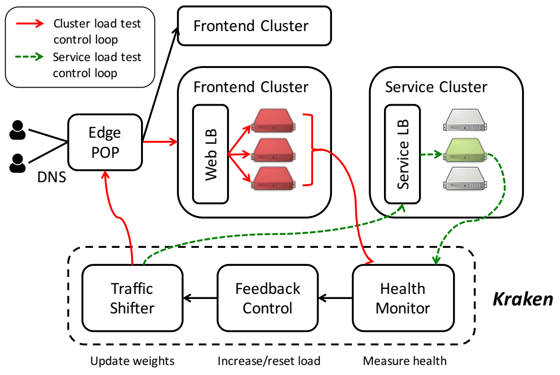


Figure 3: Kraken is a framework that allows us to load test large system units, such as clusters and entire data centers. To do so, Kraken shifts traffic from POPs to different frontend clusters while monitoring various health metrics to ensure they do not exceed allowable levels. The solid red line shows this control loop. In addition, Kraken can manage traffic in a more fine-grained manner to load test individual services composed of sets of servers, shown by the dotted green line.

gauge the true capacity of the system. If we prioritized all systems equally and tried to ensure that every system operated within its ideal performance or reliability envelope, our focus would shift to constantly tuning individual systems rather than the overall user experience. This would hurt our ability to identify the real bottlenecks to system capacity, and instead give us the infeasible challenge of improving hundreds of systems at once.

Our insight was that Kraken running on a data center is equivalent to an operational issue affecting the site—in both cases our goal is to provide a good user experience. We use two metrics, the web servers' 99th percentile response time and HTTP fatal error rate, as proxies for the user experience, and determined in most cases this was adequate to avoid bad outcomes. Over time, we have added other metrics to improve safety such as the median queuing delay on web servers, the 99th percentile CPU utilization on cache machines, etc. Each metric has an explicit threshold demarcating the vitality of the system's health. Kraken stops the test when any metric reaches its limit, before the system becomes unhealthy.

3.3 Putting it all together: Kraken

Kraken employs a feedback loop where the traffic shifting module queries Gorilla for system health before determining the next traffic shift to the system under test. As Figure 3 shows, Kraken can shift traffic onto one or more frontend clusters by manipulating the edge and cluster weights. Notice that the generic nature of Kraken also allows it to function as a framework that can be applied to any subset of the overall system.

3.4 Capacity measurement methodology

Web servers in frontend clusters are the largest component of the infrastructure. Hence, a resource bottleneck such as a performance regression or a load imbalance in a smaller subsystem that limits the throughput of the web servers is highly undesirable. We can use Kraken to identify the peak utilization a single web server can achieve. A single web server is incapable of saturating the network or any backend services, so obtaining true capacity is not difficult. This empirical web server capacity multiplied by the number of web servers in a frontend cluster yields us a *theoretical frontend cluster capacity*.

At Facebook, we have found taking this theoretical number as truth does not yield good results. As we move to larger groups of webservers, complex system effects begin to dominate and our estimates tend to miss the true capacity, sometimes by a wide margin. Kraken allows us to continually run load tests on frontend clusters to obtain the empirical frontend cluster capacity and measure the deviation from the theoretical limit. We set ourselves the aggressive goal of operating our frontend clusters at (1) 93% of their theoretical capacity limits and (2) below a target pre-defined latency threshold, while concurrently keeping pace with product evolution, user growth and frequent software release cycles. On a less frequent basis, this methodology is also applied to larger units of capacity (regions comprising multiple data centers) with a similarly aggressive goal of 90%.

3.5 Identifying and fixing bottlenecks

We have learned that running live traffic load tests without compromising on system health is difficult. Succeeding at this approach has required us to invest heavily in instrumenting our software systems, using and building new debugging tools, and encouraging engineers to collaborate on investigating and resolving issues.

Extensive data collection allows us to debug problems in situ during a test and iterate quickly. We encourage systems to leverage Scuba [1], an in-memory database that supports storing arbitrary values and querying them in real time. If a system displays a non-linear response or other unexpected behavior during a test, an engineer on call for the system can alert us and we can debug the problem using Scuba data. We use standard tools like `perf` as well as custom tracing and visualization tools [14] in our debugging.

Initially, when we identified bottlenecks we mitigated them by allocating additional capacity to the failing system. In addition to being costly, we started encountering more difficult problems, such as load imbalance and network saturation, where adding capacity had diminishing returns. We would sometimes see cases where a single shard of a backend system got orders of magnitude more traffic than its peers, causing system saturation that adding capacity could not resolve. As we developed expertise, we arrived at a more sustainable approach that includes system reconfiguration, creating and deploying new load balancing algorithms, performance tuning, and, in rare cases, system redesign. We verify the efficacy of these solutions in subsequent tests and keep iterating so we can keep pace with Facebook's evolving workload.

4 Implementation

We next describe the implementation of Kraken and how it responds to various faults.

4.1 Traffic shifting module

At Facebook, we run Proxygen, an open source software L4 and L7 load balancer. Rather than rely on a static configuration, Proxygen running on L4 load balancers in a POP reads configuration files from a distributed configuration store [41]. This configuration file lists customized edge and cluster weights for each POP as shown in Figure 2. Proxygen uses these weights to determine the fraction of user traffic to direct at each frontend cluster. By adjusting cluster weights, we can increase the relative fraction of traffic a cluster receives compared to its peers. Using edge weights we can perform the same adjustment for regions.

Kraken takes as input the target of the test and then updates the routing file stored in the configuration store with this change. The configuration store notifies the Proxygen load balancers in a remote POP of the existence of the new configuration file. In practice, we've found that it can take up to 60 seconds for Kraken to update weights, the updated configuration to be delivered to the POP, and for Proxygen to execute the requested traffic shift. Since the monitoring system, Gorilla, aggregates metrics in 60 second intervals, it takes about 120 seconds end-to-end for Kraken to initiate a traffic shift and then verify the impact of that change on system load and health.

4.2 Health monitoring module

The health monitoring system receives as input the system being tested. It then queries Gorilla [33] for metrics that can be compared to their thresholds. Gorilla stores metrics as tuples consisting of ⟨`entity`, `key`, `value`, `timestamp`⟩. These tuples are either readily available or aggregated once a minute. Thus, Kraken performs traffic shifting decisions only after waiting 60 seconds from the previous shift, but keeps querying the health monitoring system continuously to quickly react to any changes in health. Timestamps are only used to provide a monotonically increasing ordering for the data from each server; clocks do not have to be synchronized.

Table 1 provides some examples of the metrics the health monitoring module considers when judging the

Service type	Metrics
Web servers	CPU utilization, latency, error rate, fraction of operational servers
Aggregator–leaf	CPU utilization, error rate, response quality
Proxygen [39]	CPU utilization, latency, connections, retransmit rate, ethernet utilization, memory capacity utilization
Memcache [31]	Latency, object lease count
TAO [10]	CPU utilization, write success rate, read latency
Batch processor	Queue length, exception rate
Logging [23]	Error rate
Search	CPU utilization
Service discovery	CPU utilization
Message delivery	CPU utilization

Table 1: Health metrics for various systems that are affected by web load.

health of various systems. These systems are all significantly impacted by user traffic and are bellwethers of non-linear behavior. Note that the metrics in Table 1 are not intended to be comprehensive. For example, CPU utilization is the only health metric for several services, but as these services add more features and establish different constraints on performance and quality over time, other metrics may also become important in gauging their health. Observe that we monitor multiple metrics from lower-level systems like TAO, Memcache, and the Proxygen load balancers as they have a large fan-out and are critical to the health of higher-level systems.

We store the health metric definitions in a distributed configuration management system [41]. Figure 4 shows the definition for web service health in terms of error rate. We use five levels of severity when reporting metric health. `level_ranges` define the range of values for each of these levels. Metric values are sampled over the amount of time specified by `time_window`. To ensure high confidence in our assessment, we also stipulate that at least `sample_fraction` of the data points reside above a level range. We use the highest level with at least `sample_fraction` data points above that level range. If we do not receive samples, the health metric is marked as unavailable and the service is considered unhealthy. We define each of the health metrics in Table 1 in this way.

4.3 Feedback control

At the start of a test, Kraken aggressively increases load and maintains the step size while the system is healthy. We have observed a trade-off between *the rate of load increase* and *system health*. For systems that employ caching, rapid shifts in load can lead to large cache miss rates and lower system health than slow increases in load. In practice, we find that initial load increase increments of around 15% strike a good balance between load test speed and system health.

```
web_error_rate = {
  entity = 'cluster1.web',
  key = 'error.rate',
  level_ranges = [
    {BOLD => (0.0, 0.00035)},
    {MODERATE => (0.00035, 0.0004)},
    {CAUTIOUS => (0.0004, 0.00045)},
    {NOMORE => (0.00045, 0.0005)},
    {BACKOF => (0.0005, 0.001)},
  ],
  time_window = '4m',
  sample_fraction = 0.4
}
```

Figure 4: The health metric definition for web error rate.

As health metrics approach their thresholds, Kraken dynamically reduces the magnitude of traffic shifts to prevent the system from becoming overloaded. For example, when any health metric is within 10% of its threshold value, Kraken will decrease load increments to 1%. This behavior has the benefit of also allowing us to collect more precise capacity information at high load.

4.4 Handling external conditions

Kraken's feedback loop makes it responsive to any event that causes a system being load tested to be unhealthy, whether or not it was anticipated prior to the test. We have found that the infrastructure is robust enough for this mechanism alone to mitigate the majority of unexpected failures. For extreme events, we have a small set of additional remediations as described below:

- **Request spike.** Facebook experiences request spikes due to natural phenomena, national holidays, and social events, for example, we have experienced a 100% load increase in our News Feed system in the course of 30 seconds during the Super Bowl. Our simple mitigation strategy is to configure Kraken to not run a load test during national holidays or planned social and sporting events. In the event of unexpected spikes, Kraken will abort any running load tests and also distribute load to all data centers by explicitly controlling the routing configuration file published to POPs.
- **Major faults in system operation.** Sometimes, a critical low-level system might get overloaded or might have a significant reliability bug, such as a kernel crash, that is triggered during a load test. If

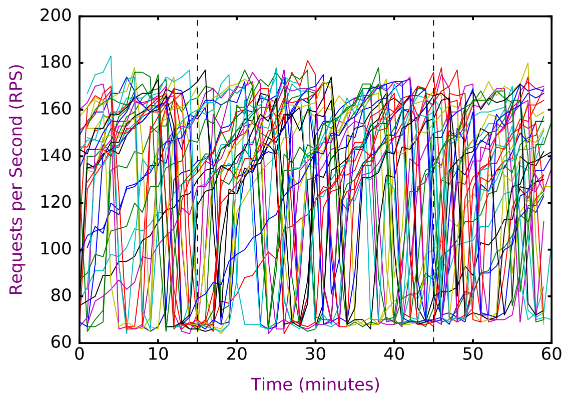


Figure 5: To measure web server capacity accurately, we continuously load test 32 web servers.

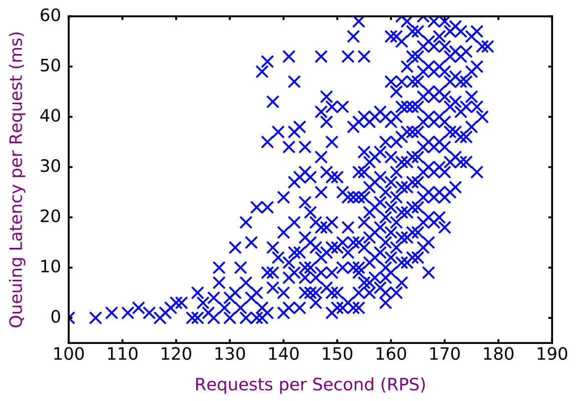


Figure 6: This plot displays the variance in the raw data from 32 load tested web servers.

this occurs, system health is bound to degrade significantly. Kraken polls Gorilla for system metrics every minute. If a fault is detected, Kraken immediately aborts the test and directs POPs to distribute load back to healthy clusters. We have recovered numerous times from issues of this sort, each time in about 2 minutes as it takes Kraken 60 seconds to detect the fault and then about 60 seconds to deliver an updated configuration to the POPs.

- **External faults such as a network partition and power loss.** As in the case above, Kraken will detect the problem and offload user traffic to other data centers in 2 minutes. If the cluster or region being tested does not recover, Kraken will decrease the load to 0, which will drain the target of all user traffic.

5 Evaluation

Kraken has been in use at Facebook for over three years and has run thousands of production load tests. As we mentioned in Section 4.2, we have augmented the set of metrics that Kraken monitors beyond the initial two, user perceivable latency and server error rate, to tens of other metrics that track the health and performance of our critical systems (cf. Table 1).

Further, we have developed a methodology around Kraken load tests that allows us to identify and resolve blockers limiting system utilization. By maintaining the pace and regularity of large scale load tests, we have incentivized teams to build instrumentation and tooling around collecting detailed data on system behavior under load. When a blocker is identified, Kraken's large scale load tests provide a structured mechanism for iteration so teams can experiment with different ideas for resolving the bottleneck and also continually improve their system performance and utilization.

Our evaluation answers the following questions:

1. Does Kraken allow us to validate capacity measurements at various scales?
2. Does Kraken provide a useful methodology for increasing utilization?

5.1 Does Kraken allow us to validate capacity measurements at various scales?

We evaluate this claim by first examining how Kraken allows us to measure the capacity of an individual server despite a continually changing workload, and then demonstrating how Kraken allows us to measure cluster and regional capacity.

5.1.1 Measuring an individual web server's capacity

Apache JMeter [4] and other load testing tools are widely employed to evaluate the capacity of an individual web server. While existing systems use a synthetic workload, Kraken directs live user traffic to measure system capacity for complex and continually changing workloads. In keeping with existing load testing systems, when measuring the capacity of an individual web server in isolation, Kraken assumes that all backend services and the network are infinite in size and cannot be saturated.

Each cluster of web servers might run a different generation of server hardware and experience a different composition of requests, so we need to run tests on web servers in each production cluster to obtain the baseline capacity for that cluster. As in a cluster load test, we use a preset error rate and latency metric as thresholds for when the system is operating at peak capacity. However, as we are testing a less complex system, we can be more exact. We use queuing latency on the server as our primary metric to gauge capacity—if the server begins to queue consistently, it is no longer able to keep up with the workload we are sending to it, and we have reached the individual server's max capacity.

We found that using a single server results in too much variance in the output capacity number. Through experi-

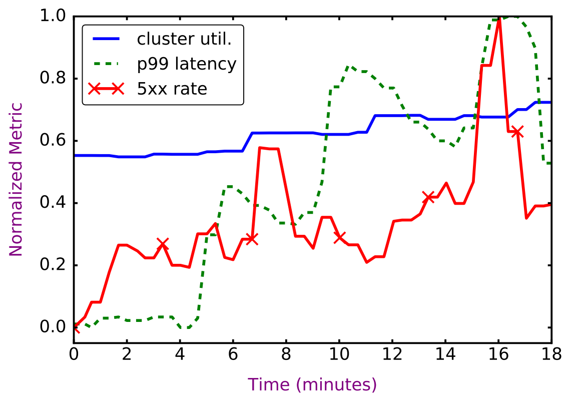
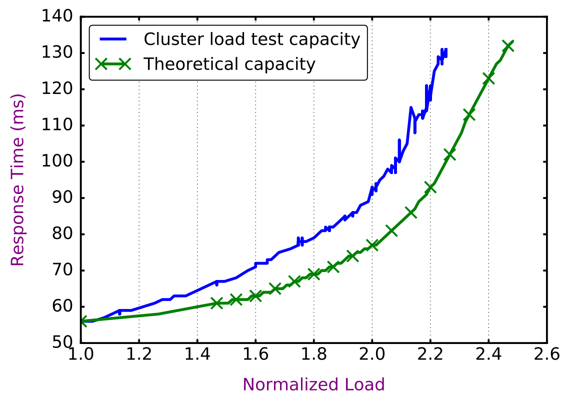
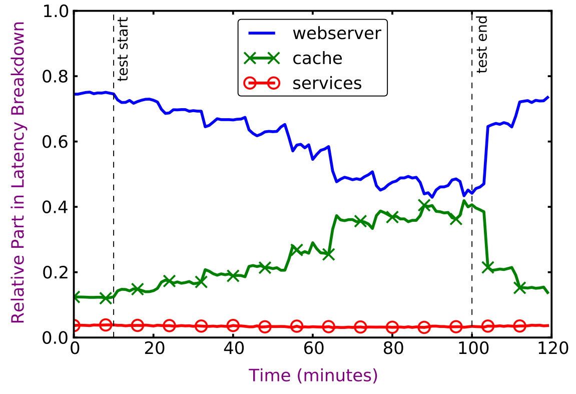


Figure 7: Kraken runs a cluster load test by directing increasing amounts of user traffic at a cluster. A load test affects both the cluster's CPU utilization and health metrics such as the HTTP 5xx error rate and response latency.



(a) Performance gap between cluster load test capacity and theoretical capacity.



(b) Latency breakdown in cluster load test.

Figure 8: (a) Demonstrates the performance gap between the cluster load test capacity and theoretical capacity. (b) Shows the latency breakdown that allows us to identify which of the caching systems or miscellaneous services contribute most to the performance gap. We observe an increase in time spent waiting for cache response as the cluster load increases, indicating that cache is the bottleneck.

mentation, we have found 32 servers to be an ideal number for our workload. Further increases in machine count do not significantly reduce variance. Figure 5 depicts a load test run on 32 independent web servers in a 60 minute interval. At the time this test was performed the servers were able to perform about 175 requests per second before the queuing latency threshold was reached.

Figure 6 shows the data points collected in a 30 minute window during the load test shown in Figure 5. Each data point plots the requests per second against CPU delay per request in milliseconds for a single web server, averaged over one minute. Our initial load test was run with a relaxed threshold for the sake of illustration. Once queuing begins, increases in allowed queuing result in quickly diminishing returns on throughput. In practice, we apply a more conservative limit of 20 ms queuing delay. To get our baseline server capacity, we simply take the average of the 32 servers. We use pick-2 load balancing [28] to ensure we evenly utilize servers, so we do not need to worry about the variance in hardware between servers. We can then derive the theoretical cluster capacity by multiplying the per-server capacity by the number of servers in a cluster.

5.1.2 Measuring a cluster's capacity

Figure 7 shows the execution of a cluster load test with Kraken. The line labeled `cluster util.` shows the current requests per second coming in to the cluster normalized by the cluster's theoretical max requests per second. Kraken initiates the cluster load test at minute 0 and the test concludes at minute 18. Every 5 minutes, Kraken inspects the health of the cluster and makes a decision for how to shift traffic. As described in Section 4.1, it takes Kraken about 2 minutes to execute a load shift, which is evident as cluster utilization changes around minute 7 for a decision made at minute 5. Notice that as the test proceeds, the cluster's health begins to degrade so Kraken decreases the magnitude of the traffic shifts until a peak utilization of 75% is hit. Kraken resets the load of the cluster in two traffic shifts over the course of 10 minutes (not shown in Figure 7).

Kraken closely monitors the health of the system when running this load test. The lines labeled `p99 latency` and `5xx rate` in Figure 7 correspond to the two initial health metrics we monitor: user perceivable latency and server error rate, respectively. Both of these metrics are normalized to the minimum and maximum values measured during the test. The three spikes in latency are a direct result of Kraken directing new users at this cluster—requests from the new users cause cache misses and require new data to be fetched. This test stopped due to the p99 latency, which is initially low but sustains above the threshold level for too long after the third traffic shift.

The final utilization of this cluster was 75% of the theoretical maximal. This is below our target utilization of 93% of the theoretical maximum, and we consider this load test unsuccessful. We next turn to how we identify and fix the issues that prevent a cluster from getting close to its theoretical maximal utilization.

Why did the cluster not hit its theoretical max utilization? Figure 8(a) depicts a different load test that hit a latency threshold. Here, the cluster performance (solid line) diverges from the theoretical web server performance (crossed line). To identify the utilization bottleneck, we drill into the data for other subsystems involved in serving a web request: the web server, cache, and other services. Figure 8(b) breaks down the web server response time between these three components. As load increases, the proportion of web server latency (the unmarked line) decreases while the proportion of cache latency (the crossed line) increases from around 15% at the point labeled `test start` to around 40% at the point labeled `test end` while service latency (the ○ line) remains unaffected, identifying cache as the bottleneck.

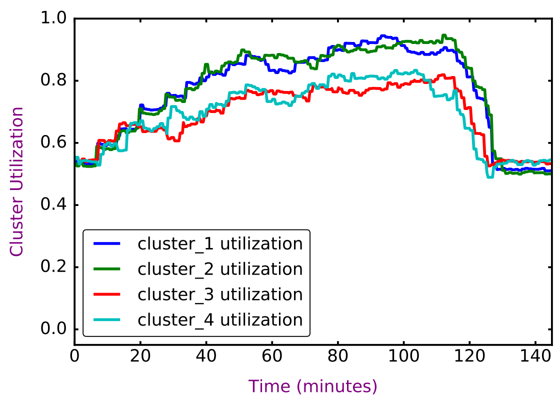
5.1.3 Measuring a region's capacity

Figure 9(a) shows that we can use Kraken to direct user traffic at multiple clusters simultaneously in a regional load test to stress systems that span clusters. Kraken reacts to variances in system health (for example, 5xx error rate, shown in Figure 9(b)) by decreasing user traffic to the tested clusters. Kraken maintains high load at about 90% utilization for about an hour—we intentionally hold the region under high load for an extended period of time to allow latent effects to surface. Kraken stops the test at minute 111 and quickly resets the load to normal levels in 15 minutes.

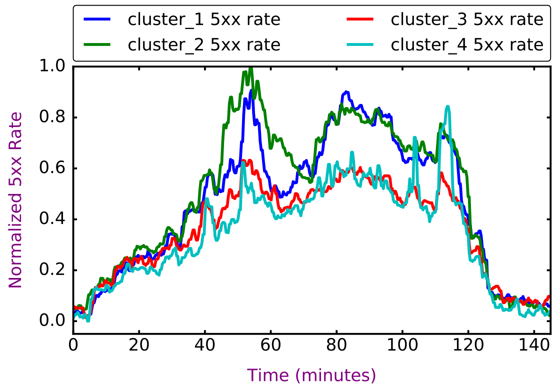
5.2 Does Kraken provide a useful methodology for increasing utilization?

Figure 10 depicts a load test from May 2015 where one of Kraken's regional load tests hit 75% utilization before encountering bottlenecks. Test outcomes of this form were the norm in 2014 and early 2015 as we were still developing the Kraken methodology for identifying and resolving bottlenecks encountered in cluster and regional load tests. Figure 9 depicts our current status where our regional load tests almost always hit their target utilization of 90% theoretical max resulting in a 20% system utilization improvement.

In this section, we describe how Kraken allowed us to surface many bottlenecks that were hidden until the systems were under load. We identified problems, experimented with remedies, and iterated on our solutions over successive tests. Further, this process of continually testing and fixing allowed us to develop a library of solutions and verify health without permitting regressions.



(a) Cluster utilization



(b) Cluster 5xx error rate

Figure 9: Kraken measures the capacity of a region by directing user traffic to all of the frontend clusters in the region simultaneously. (a) shows the effects on the data center clusters' utilization. (b) shows how the 5xx rate of the clusters changed during this time. Notice that when a cluster's HTTP fatals (5xx errors) increase, Kraken reduces the load on the cluster so it operates within a healthy range.

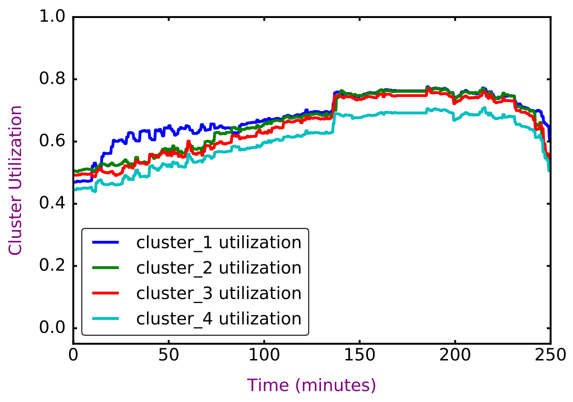


Figure 10: In this May 2015 test, Kraken pushes the frontend clusters in a region to an average of 75.7% utilization of their theoretical max before hitting pre-set thresholds. This load test was unsuccessful.

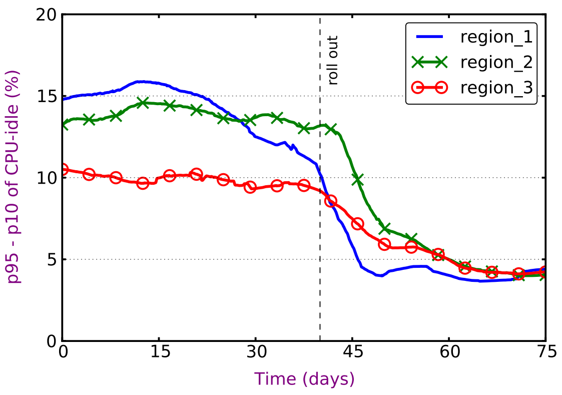


Figure 11: The spread in CPU utilization as measured by the difference between the 95th and 10th percentile CPU utilization in a cluster in different geographic regions. After using Kraken to identify a bottleneck in the shared cache resource, we deployed a technique to alleviate the bottleneck, resulting in a lower CPU utilization spread.

5.2.1 Hash weights for cache

The cache bottleneck depicted in Figure 8(b) was the first major issue that Kraken helped identify and resolve.

How did Kraken expose the cache bottleneck? Several cluster load tests were blocked by latency increases. Further inspection revealed that during the load test there was a disproportionate increase in the fraction of time spent retrieving data from TAO [10], a write-through cache that stores heavily accessed data. We engaged with the TAO engineers and after some instrumentation, learned that the latency increase was due to just a few cache machines that had significantly higher than average load during the test. It took several load tests to gather sufficient data to diagnose the bottleneck.

What was the cache bottleneck? TAO scales by logically partitioning its data into shards and randomly assigning these shards to individual cache servers. When constructing the response to a request, the web server might have to issue multiple rounds of data fetches, which might access hundreds of cache servers. TAO's hashing algorithm had been designed 4 years prior to the test. At that time, the ratio of shards to TAO servers was larger, and the shards were more uniform in request rate. Over time these assumptions changed, causing imbalance among the servers. Kraken's cluster load tests revealed that a small number of cache servers were driven out of CPU because they stored a significant fraction of all frequently-accessed data (e.g., popular content); this adversely affected all web servers that accessed that data. Instead of assigning shards to cache servers with a uniform hash, the solution was to assign each server a tunable *hash weight* based on the frequency of access, giving us finer control to even out the distribution of shards and balance load.

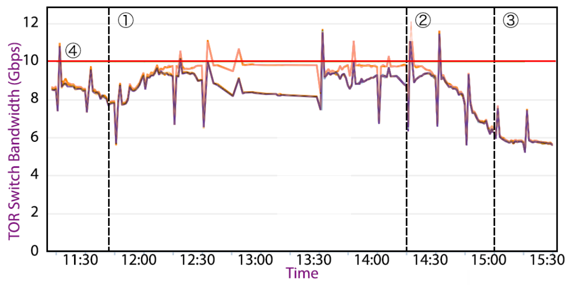


Figure 12: A spike in network traffic during a Kraken load test saturates two top-of-rack network switches. Alleviating this issue required relocating services running in the racks.

How did we validate the cache bottleneck fix? We leveraged Kraken to run multiple cluster load tests in different regions to validate the fix. Figure 11 depicts the outcome of hash weights—the x-axis shows time in days and the y-axis is the *difference* between the 95th and 10th percentile CPU utilization (i.e., the CPU utilization *spread* between highly-loaded cache servers and lowly-loaded cache servers) across the cache servers in clusters spread among three regions. Notice that after hash weights were rolled out, the spread in CPU utilization was reduced from an initial 10–15% to less than 5%. In tests following the fix, we were able to verify that this solution maintained its effectiveness at high load, and that we were able to apply more load to the clusters being tested. This solution addressed our short term capacity bottleneck and also significantly improved the reliability and scalability of TAO.

5.2.2 Network saturation

Most of Kraken's load tests in its first year of production were blocked by issues and bottlenecks in our higher-level software systems. As these bottlenecks were resolved, latent system issues at lower levels of the stack were revealed. During one test, which failed due to high latency, the latency effect was attributed to several disconnected services rathan than a single service. Further analysis revealed that the effect was localized to particular data center racks, which are often composed of machines from different services.

Figure 12 depicts the bandwidth utilization of two top-of-rack network switches during a Kraken load test. In this figure (and in the other case study figures in this section), we have labeled the phases of a load test: ① load test begins, ② load test starts decreasing load, and ③ load test ends. The nominal switch bandwidth limit of 10 Gbps, beyond which retransmissions or packet loss may occur, has been labeled ④ (specific to Figure 12).

Notice that the load test results in rack switches experiencing higher load due to the additional requests and

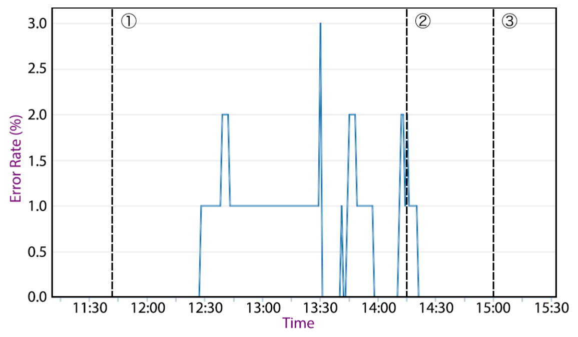


Figure 13: Saturated top-of-rack network switches result in services in the rack experiencing a 3% error rate during a Kraken load test.

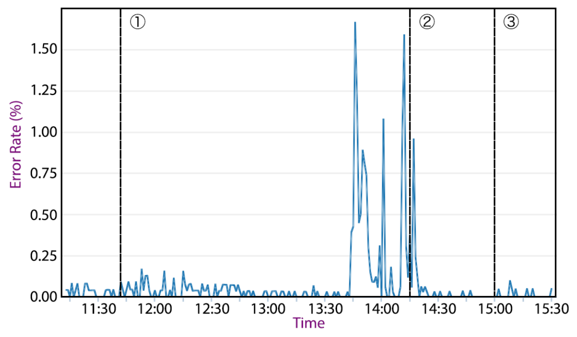


Figure 14: A Kraken load test triggers a 1.75% error rate in a critical classification service.

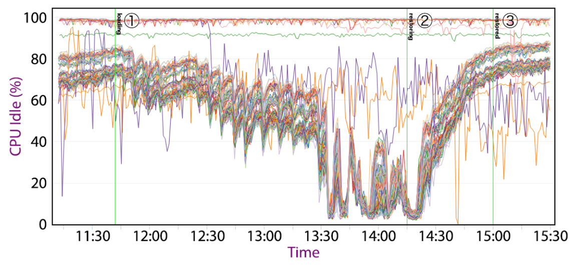


Figure 15: The multiple bands of CPU utilization clearly reveal a load imbalance in the machines running the classification service. This imbalance worsens under load and becomes a bottleneck that prevents us from fully utilizing the data center.

responses being sent to/from servers located in the racks, with peak bandwidth nearing 12 Gbps. These bandwidth spikes resulted in response errors, shown in Figure 13. Notice that around 13:30, nearly 3% of all requests experience an error.

An investigation revealed that these two racks housed a large number of machines belonging to a high-volume storage service that is used to retrieve posts when a user loads News Feed. During the load test, these machines received a higher volume of requests than normal due to the increased user traffic directed at this cluster and their responses saturated the uplink bandwidth of the top-of-rack network switch. We mitigated this problem by distributing this service's machines evenly across multiple racks in the data center.

Prior to surfacing this issue, we were aware of network saturation as a potential issue, but did not anticipate it as a bottleneck. This demonstrated the ability of Kraken to identify unexpected bottlenecks in the system. As a result we built a new network monitoring tool that identifies network utilization bottlenecks during Kraken's load tests and alerts service developers.

5.2.3 Poor load balancing

During a Kraken load test, a critical classification service in the data center under test experienced a 1.75% error rate in serving its requests (Figure 14). Note that this error rate is considered too high for the service.

Investigating uncovered the fact that the load was not evenly distributed among instances of the service as shown in Figure 15. Note that this load imbalance is clearly visible at time 11:30 (before the test commences) but it is only at time 13:30 that it is evident that imbalanced load imperils the service, as some fraction of machines are out of CPU. We can also observe an oscillation in CPU utilization between 13:30 and 14:15, due to the process hitting a failure condition because it is unable to properly handle the excess load. The over-utilization of a small handful of machines bottlenecks the entire region and results in a degraded performance.

In this case, pick-2 load balancing [28] was applied to even the load between servers. Failing under heavy load was also a suboptimal behavior we uncovered. For some services, techniques such as adaptive load shedding [17, 48] can be effective at reducing resource load while still servicing requests. In the case of this service, failures are difficult to tolerate, so a proportional-integral-derivative controller [37] was implemented to help continue to serve requests under high load.

5.2.4 Misconfiguration

Figure 16 shows an example of Kraken's health monitoring. One of the metrics that Kraken monitors is the error rate of a story ranking and delivery service. Recall that we have defined thresholds corresponding to how healthy a metric is: `MODERATE` (labeled A), `CAUTIOUS` (labeled B), `NOMORE` (labeled C), and `BACKOFF` (labeled D) (note that `BOLD` is not labeled as it corresponds to the range below `MODERATE`). Kraken uses the indicated thresholds for this health metric when making load change decisions.

We then worked with the ranking service's developers to instrument their system. Figure 17 shows that the service was experiencing multiple types of errors under load, with the most severe impact from "connection gating" where the number of concurrent threads requesting

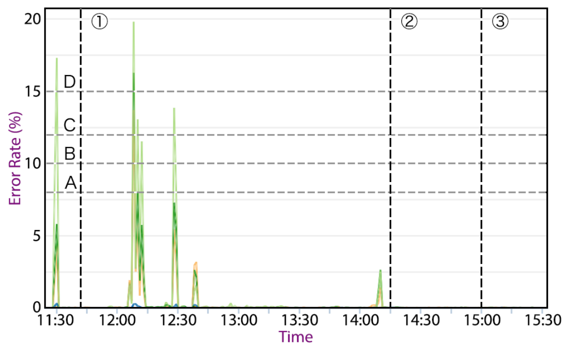


Figure 16: A load test triggers error spikes in a misconfigured service.

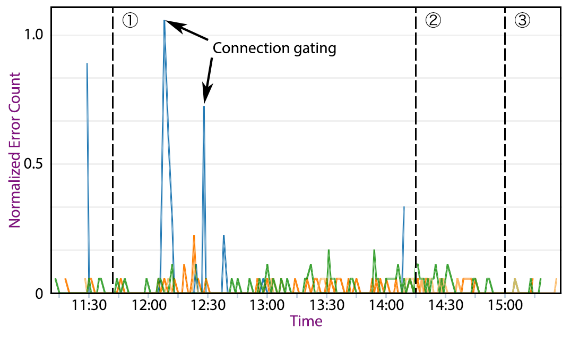


Figure 17: Failed responses in a misconfigured service.

data from an external database was limited to a fixed amount. While the intent of connection gating is to avoid exacerbating the effects of requesting data from a slow database instance, the actual gating value configured for this service was too low for the request rate it was servicing.

This bottleneck was resolved by increasing the number of concurrent threads allowed to request data from the database, and then running a follow-up Kraken load test to verify that no failed responses were returned by the ranking service, and that we could utilize the region more effectively as a result.

5.2.5 Insufficient capacity

Well-tuned services without any hardware or software bottlenecks may still lack sufficient capacity. This issue arises due to organic increases in service usage that are sometimes difficult for service developers to predict.

One surprising revelation was a Kraken load test that exposed a capacity issue in Gorilla [33], the service that Kraken relies on for monitoring and data collection. During a routine regional load test, Gorilla started experiencing increasing queuing, ultimately causing it to drop some requests as they began timing out. Figure 18 depicts the decrease in Enqueued requests and the increase in Dropped requests. Kraken aborted the load test as its monitoring component registered incomplete metrics.

We followed up by analyzing Gorilla. It turned out that the data center had recently acquired a new frontend cluster. As a result, when Kraken ran a regional load test, a much larger set of users were directed at the region and overloaded the existing Gorilla machines. The link between web traffic and Gorilla traffic is indirect, as Gorilla is accessed primarily by services rather than the web servers themselves, so the developers of Gorilla did not realize additional capacity would be required.

The mitigation was to allocate additional capacity to Gorilla to keep pace with increased demand. Note that these capacity allocations do not impact the service's efficiency requirements, which are benchmarked by regular load tests to ensure that performance regressions do not creep in and decrease throughput under high load.

5.3 Service-level load testing

Regional load tests are effective at finding service-level issues. However, service regressions and misconfigurations can be identified without requiring large scale tests, whereas problems like TAO load imbalance (see Section 5.2.1) are system effects that only manifest as bottlenecks at scale. Providing service owners a way to test their systems independently allows us to focus on identifying bottlenecks due to cross-system dependencies during regional tests.

Encouraged by Kraken's utility in characterizing system performance, we extended it to support load tests on individual services. When testing a service, Kraken dynamically updates the service's load balancing configuration to gradually concentrate traffic onto the designated set of target machines. Kraken monitors service-specific metrics that track resource usage, latency, and errors and measures both the service capacity and its behavior under load. Figure 3 illustrates how Kraken performs a service-level load test.

The goal of service-level load tests is to enable developers to quickly identify performance issues in their services without needing to wait until the next regional test. In addition to identifying regressions, resource exhaustion, and other utilization bottlenecks, service-specific load tests allow developers to evaluate different optimizations in production on a subset of their fleet.

For example, Figure 19 shows a load test performed on a set of News Feed's ranking aggregators. Load is measured in queries per second (QPS). As the test proceeds, QPS increases and the fraction of idle CPU time decreases. Notice that when the fraction of idle CPU time reaches 25%, the News Feed aggregator dynamically adjusts result ranking complexity. This ensures that the News Feed aggregator can continue to serve all incoming requests without exhausting CPU resources. This continues over the range of load labeled QoS control until,

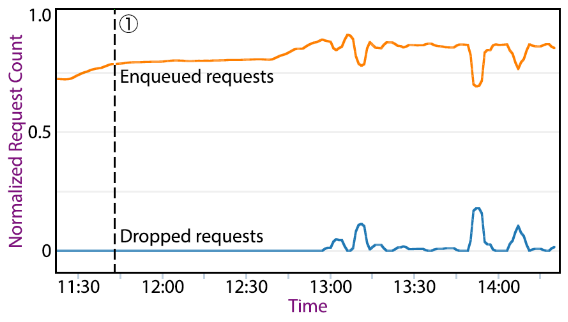


Figure 18: Gorilla drops requests due to insufficient capacity.

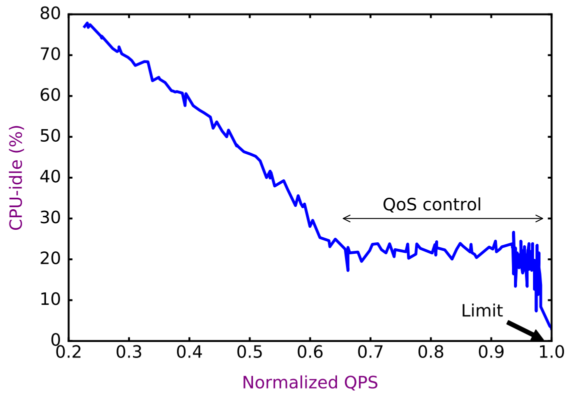


Figure 19: Kraken load testing News Feed. The range labeled QoS control shows the News Feed aggregators dynamically adjusting request quality to be able to serve more requests.

eventually, the fraction of CPU idle time drops below a pre-determined threshold and the load test ends.

6 Related Work

Large scale web services such as Amazon, Facebook, Google, Microsoft, and others serve a global audience that expect these services to be highly available. To ensure high availability in the face of disasters, these services often operate out of multiple geo-replicated data centers [3, 21]. Additionally, these services rely on modeling for capacity planning and resource allocation.

Capacity management based on theoretical system modeling is a widely studied topic. Some recent works in this field [20, 24, 46, 51], recognize the challenges of modeling multi-tier systems with changing workloads and propose various schemes for dynamic workload-driven adjustments to the theoretical allocations. There is an understanding that large scale distributed systems often times demonstrate *emergent behavior*, which cannot be predicted through analysis at any level simpler than that of the system as a whole [22, 29, 43]. These prior systems support Kraken's motivation for load testing to glean empirical data for dynamic capacity management.

Many prior systems such as Muse [11] and others [5, 32, 38, 47] automate resource allocation using instrumentation to provide prior history, offline and online analysis or simple feedback control to observe the system under testing, and iteratively apply an optimization policy until their objective function is satisfied. Doyle et. al. [18] build on the Muse system and propose a model-based approach to provisioning resources. Other systems utilize queuing [24, 30, 40] or energy [13, 19] models to predict how large systems will behave under load and how much power they will consume, to aid capacity management or maximize resource utilization.

Rather than derive analytical models for managing data centers, several recent systems propose experimentation on production systems [6, 50]. JustRunIt [50] proposes the use of sandboxed deployments that can execute shadow traffic from a real world deployment to answer various "what-if" questions. We make the same observation as JustRunIt but take their idea further as we wish to evaluate the performance and capacity limits of a non-virtualized production deployment.

Our strategy is to leverage load testing to evaluate and manage our systems. There are many open source and commercial load testing systems including Apache JMeter [4] and its variants such as BlazeMeter [9] as well as alternate frameworks such as Loader [25] that provide load testing as a service. We were unable to leverage these tools in our work, as they were limited in their scope. For instance, JMeter does not execute JavaScript while Loader only simulates connections to a web service. Instead, Kraken allows us to load test production systems with live user traffic.

Tang et. al. [42] leverage load testing to profile NUMA usage at Google but do not describe how their technique can be applied to identify higher-level bottlenecks or resource misallocation.

Kraken's analytics share ideas with the deeply developed field of performance analysis, which has always been crucial for detecting regressions and discovering bottlenecks in large scale distributed systems. While some previous performance analysis systems leverage fine-grained instrumentation such as Spectroscope [36], Magpie [7] and Pinpoint [12], others rely on passive analytics, investigating network flows [2], or logging information [14]. Kraken infers the dominant causal dependencies and bottlenecks by relying on the existing monitoring mechanisms provided by individual services. This allows for flexibility in the analysis of large heterogeneous systems. Various machine learning and statistics techniques have been developed for improving performance analysis of distributed systems [15, 16, 26]; Kraken's algorithms are much simpler so that its operations can be easily reasoned about by a human.

Kraken is not the first large scale test framework that works with live user traffic. Netflix's Chaos Monkey [8, 45] induces failure in some component of an Amazon Web Services deployment and empirically confirms that the service is capable of tolerating faults and degrading gracefully under duress.

7 Experience

Since deploying Kraken to production, we have run over 50 regional load tests and thousands of cluster load tests. We list some lessons learned below:

- Generating capacity and utilization models with a continually changing workload is difficult. A competing approach, empirical testing, is a simpler alternative.
- Simplicity is key to Kraken's success. When load causes multiple systems to fail in unexpected ways, we need the stability of simple systems to debug complex issues.
- Identifying the right metrics that capture a complex system's performance, error rate, and latency is difficult. We have found it useful to identify several candidate metrics and then observe their behavior over tens to hundreds of tests to determine which ones provide the highest signal. However, once we identify stable metrics, their thresholds are easy to configure and almost never change once set.
- We find that specialized error handling mechanisms such as automatic failover and fallback mechanism can make systems harder to debug and lead to unexpected costs. These mechanisms have the ability to hide problems without resolving root causes, often leading small problems to snowball into bigger issues before detection and resolution. We find that such mitigations need to be well-instrumented to be effective in the long run, and prefer more direct methods such as graceful degradation.
- Bottleneck resolutions such as allocating capacity to needy services, changing system configuration or selecting different load balancing strategies have been critical for fixing production issues fast. We turn to heavy-weight resolutions like profiling, performance tuning, and system redesign only if the benefit justifies the engineering and capacity cost.
- While some systems prefer running on non-standard hardware or prefer non-uniform deployments in data centers, we have found that trading off some amount of efficiency and performance for simplicity makes systems much easier to operate and debug.

8 Conclusion

Large scale web services are difficult to accurately model because they are composed of hundreds of rapidly evolving software systems, are distributed across geo-replicated data centers, and have constantly changing workloads. We propose Kraken, a system that leverages live user traffic to empirically load test every level of the infrastructure stack to measure capacity. Further, we describe a methodology for identifying bottlenecks and iterating over solutions with successive Kraken load tests to continually improve infrastructure utilization. Kraken has been in production for the past three years. In that time, Kraken has run thousands of load tests and allowed us to increase Facebook's capacity to serve users by over 20% using the same hardware.

9 Acknowledgments

We thank the anonymous reviewers and our shepherd, Rebecca Isaacs, for comments that greatly improved this paper. We also thank Théo Dubourg, Constantin Dumitrascu, David Felty, Calvin Fu, Yun Jin, Silvio Soares Ribeiro Junior, Daniel Peek, and the numerous developers at Facebook who have given us feedback on the tooling, monitoring, and methodology of Kraken and helped us improve our infrastructure utilization.

References

[1] ABRAHAM, L., ALLEN, J., BARYKIN, O., BORKAR, V., CHOPRA, B., GEREA, C., MERL, D., METZLER, J., REISS, D., SUBRAMANIAN, S., WIENER, J., AND ZED, O. Scuba: Diving into data at Facebook. In *Proceedings of the 39th International Conference on Very Large Data Bases* (August 2013), VLDB '13.

[2] AGUILERA, M. K., MOGUL, J. C., WIENER, J. L., REYNOLDS, P., AND MUTHITACHAROEN, A. Performance debugging for distributed systems of black boxes. In *Proceedings of the 19th ACM Symposium on Operating Systems Principles* (2003), SOSP '03.

[3] AMAZON. Regions and availability zones. `http://docs.aws.amazon.com/AWSEC2/latest/UserGuide/using-regions-availability-zones.html`.

[4] Apache JMeter. `http://jmeter.apache.org/`.

[5] ARON, M., DRUSCHEL, P., AND ZWAENEPOEL, W. Cluster reserves: A mechanism for resource management in cluster-based network servers. In *Proceedings of the 2000 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems* (2000), SIGMETRICS '00.

[6] BABU, S., BORISOV, N., DUAN, S., HERODOTOU, H., AND THUMMALA, V. Automated experiment-driven management of (database) systems. In *Proceedings of the 12th*

Conference on Hot Topics in Operating Systems (2009), HotOS '09.

[7] Barham, P., Donnelly, A., Isaacs, R., and Mortier, R. Using Magpie for request extraction and workload modelling. In *Proceedings of the 6th USENIX Symposium on Operating Systems Design and Implementation* (2004), OSDI '04.

[8] Basiri, A., Behnam, N., de Rooij, R., Hochstein, L., Kosewski, L., Reynolds, J., and Rosenthal, C. Chaos engineering. *IEEE Software 33*, 3 (May 2016), 35–41.

[9] BlazeMeter. `http://blazemeter.com/`.

[10] Bronson, N., Amsden, Z., Cabrera, G., Chakka, P., Dimov, P., Ding, H., Ferris, J., Giardullo, A., Kulkarni, S., Li, H. C., Marchukov, M., Petrov, D., Puzar, L., Song, Y. J., and Venkataramani, V. TAO: Facebook's distributed data store for the social graph. In *Proceedings of the 2013 USENIX Annual Technical Conference* (2013), USENIX ATC '13.

[11] Chase, J. S., Anderson, D. C., Thakar, P. N., Vahdat, A. M., and Doyle, R. P. Managing energy and server resources in hosting centers. In *Proceedings of the 18th ACM Symposium on Operating Systems Principles* (2001), SOSP '01.

[12] Chen, M. Y., Kiciman, E., Fratkin, E., Fox, A., and Brewer, E. Pinpoint: Problem determination in large, dynamic internet services. In *Proceedings of the 2002 International Conference on Dependable Systems and Networks* (2002), DSN '02.

[13] Chen, Y., Das, A., Qin, W., Sivasubramaniam, A., Wang, Q., and Gautam, N. Managing server energy and operational costs in hosting centers. In *Proceedings of the 2005 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems* (2005), SIGMETRICS '05.

[14] Chow, M., Meisner, D., Flinn, J., Peek, D., and Wenisch, T. F. The mystery machine: End-to-end performance analysis of large-scale internet services. In *Proceedings of the 11th USENIX Symposium on Operating Systems Design and Implementation* (2014), OSDI '14.

[15] 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 USENIX Symposium on Operating Systems Design and Implementation* (2004), OSDI '04.

[16] Cohen, I., Zhang, S., Goldszmidt, M., Symons, J., Kelly, T., and Fox, A. Capturing, indexing, clustering, and retrieving system history. In *Proceedings of the 20th ACM Symposium on Operating Systems Principles* (2005), SOSP '05.

[17] 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* (2014), SOCC '14.

[18] Doyle, R. P., Chase, J. S., Asad, O. M., Jin, W., and Vahdat, A. M. Model-based resource provisioning in a web service utility. In *Proceedings of the 4th USENIX Symposium on Internet Technologies and Systems* (2003), USITS '03.

[19] Fan, X., Weber, W.-D., and Barroso, L. A. Power provisioning for a warehouse-sized computer. In *Proceedings of the 34th Annual International Symposium on Computer Architecture* (2007), ISCA '07.

[20] Gmach, D., Rolia, J., Cherkasova, L., and Kemper, A. Resource pool management: Reactive versus proactive or let's be friends. *Computer Networks: The International Journal of Computer and Telecommunications Networking 53*, 17 (December 2009), 2905–2922.

[21] Google. Google compute engine: Regions and zones. `https://cloud.google.com/compute/docs/zones`.

[22] Gribble, S. D. Robustness in complex systems. In *Proceedings of the 8th Workshop on Hot Topics in Operating Systems* (2001), HOTOS '01.

[23] Jain, N., Borthakur, D., Murthy, R., Shao, Z., Antony, S., Thusoo, A., Sarma, J., and Liu, H. Data Warehousing and Analytics Infrastructure at Facebook. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of Data* (2010), SIGMOD '10.

[24] Liu, X., Heo, J., and Sha, L. Modeling 3-tiered web applications. In *Proceedings of the 13th IEEE International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems* (2005), MASCOTS '16.

[25] Loader: simple cloud-based load testing. `http://loader.io`.

[26] MALIK, H. A methodology to support load test analysis. In *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering* (2010), ICSE '10.

[27] MICHELSEN, M. Continuous deployment at Quora. `http://engineering.quora.com/Continuous-Deployment-at-Quora`.

[28] MITZENMACHER, M. The power of two choices in randomized load balancing. *IEEE Transactions on Parallel and Distributed Systems 12* (October 2001), 1094–1104.

[29] MOGUL, J. C. Emergent (mis)behavior vs. complex software systems. In *Proceedings of the 1st ACM SIGOPS/EuroSys European Conference on Computer Systems* (2006), EuroSys '06.

[30] NAIR, J., WIERMAN, A., AND ZWART, B. Provisioning of large-scale systems: The interplay between network effects and strategic behavior in the user base. *Management Science 62*, 6 (November 2016), 1830–1841.

[31] NISHTALA, R., FUGAL, H., GRIMM, S., KWIATKOWSKI, M., LEE, H., LI, H., 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* (2013), NSDI '13.

[32] PADALA, P., SHIN, K. G., ZHU, X., UYSAL, M., WANG, Z., SINGHAL, S., MERCHANT, A., AND SALEM, K. Adaptive control of virtualized resources in utility computing environments. In *Proceedings of the 2nd ACM SIGOPS/EuroSys European Conference on Computer Systems* (2007), EuroSys '07.

[33] PELKONEN, T., FRANKLIN, S., TELLER, J., CAVALLARO, P., HUANG, Q., MEZA, J., AND VEERARAGHAVAN, K. Gorilla: A fast, scalable, in-memory time series database. In *Proceedings of the 41st International Conference on Very Large Data Bases* (2015), VLDB '15.

[34] RIVIORE, S., SHAH, M. A., RANGANATHAN, P., AND KOZYRAKIS, C. Joulesort: A balanced energy-efficiency benchmark. In *Proceedings of the 2007 ACM SIGMOD International Conference on Management of Data* (2007), SIGMOD '07.

[35] ROSSI, C. Ship early and ship twice as often, 2012. `https://www.facebook.com/notes/facebook-engineering/ship-early-and-ship-twice-as-often/10150985860363920`.

[36] SAMBASIVAN, R. R., ZHENG, A. X., DE ROSA, M., KREVAT, E., WHITMAN, S., STROUCKEN, M., WANG, W., XU, L., AND GANGER, G. R. Diagnosing performance changes by comparing request flows. In *Proceedings of the 8th USENIX Networked Systems Design and Implementation* (2011), NSDI '11.

[37] SELLERS, D. An overview of proportional plus integral plus derivative control and suggestions for its successful application and implementation. In *Proceedings of the 1st International Conference for Enhanced Building Operations* (2001), ICEBO '01.

[38] SHEN, K., TANG, H., YANG, T., AND CHU, L. Integrated resource management for cluster-based internet services. In *Proceedings of the 5th USENIX Symposium on Operating Systems Design and implementation* (2002), OSDI '02.

[39] SOMMERMANN, D., AND FRINDELL, A. Introducing Proxygen, Facebook's C++ HTTP framework, 2014. `https://code.facebook.com/posts/1503205539947302/introducing-proxygen-facebook-s-c-http-framework/`.

[40] STEWART, C., AND SHEN, K. Performance modeling and system management for multi-component online services. In *Proceedings of the 2nd USENIX Networked Systems Design and Implementation* (2005), NSDI '05.

[41] TANG, C., KOOBURAT, T., VENKATACHALAM, P., CHANDER, A., WEN, Z., NARAYANAN, A., DOWELL, P., AND KARL, R. Holistic configuration management at Facebook. In *Proceedings of the 25th Symposium on Operating Systems Principles* (October 2015), SOSP '15.

[42] TANG, L., MARS, J., ZHANG, X., HAGMANN, R., HUNDT, R., AND TUNE, E. Optimizing Google's warehouse scale computers: The NUMA experience. In *Proceedings of the 19th International Symposium on High-Performance Computer Architecture* (2013), HPCA '13.

[43] THERESKA, E., AND GANGER, G. R. IRON-Model: Robust performance models in the wild. In *Proceedings of the 2008 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems* (2008), SIGMETRICS '08.

[44] TPC-C. `http://www.tpc.org/tpcc/`.

[45] TSEITLIN, A. The antifragile organization. *Communications of the ACM 56*, 8 (August 2013), 40–44.

[46] URGAONKAR, B., SHENOY, P., CHANDRA, A., GOYAL, P., AND WOOD, T. Agile dynamic provisioning of multi-tier internet applications. *ACM Transactions on Autonomous and Adaptive Systems (TAAS) 3*, 1 (March 2008), 1–39.

[47] URGAONKAR, B., SHENOY, P., AND ROSCOE, T. Resource overbooking and application profiling in shared hosting platforms. In *Proceedings of the 5th USENIX Symposium on Operating Systems Design and implementation* (2002), OSDI '02.

[48] WELSH, M., AND CULLER, D. Adaptive overload control for busy internet servers. In *USENIX Symposium on Internet Technologies and Systems* (2003), USITS '03.

[49] Yahoo! cloud serving benchmark (YCSB). `https://github.com/brianfrankcooper/YCSB/wiki`.

[50] ZHENG, W., BIANCHINI, R., JANAKIRAMAN, G. J., SANTOS, J. R., AND TURNER, Y. JustRunIt: Experiment-based management of virtualized data centers. In *Proceedings of the 2009 USENIX Annual Technical Conference* (2009), USENIX '09.

[51] ZHU, X., YOUNG, D., WATSON, B. J., WANG, Z., ROLIA, J., SINGHAL, S., MCKEE, B., HYSER, C., GMACH, D., GARDNER, R., CHRISTIAN, T., AND CHERKASOVA, L. 1000 islands: Integrated capacity and workload management for the next generation data center. In *Proceedings of the 2008 International Conference on Autonomic Computing* (2008), ICAC '08.

CertiKOS: An Extensible Architecture for Building Certified Concurrent OS Kernels

Ronghui Gu   Zhong Shao   Hao Chen   Xiongnan (Newman) Wu
Jieung Kim   Vilhelm Sjöberg   David Costanzo
Yale University

Abstract

Complete formal verification of a non-trivial concurrent OS kernel is widely considered a grand challenge. We present a novel compositional approach for building certified concurrent OS kernels. Concurrency allows interleaved execution of kernel/user modules across different layers of abstraction. Each such layer can have a different set of observable events. We insist on formally specifying these layers and their observable events, and then verifying each kernel module at its proper abstraction level. To support certified linking with other CPUs or threads, we prove a strong contextual refinement property for every kernel function, which states that the implementation of each such function will behave like its specification under any kernel/user context with any valid interleaving. We have successfully developed a practical concurrent OS kernel and verified its (contextual) functional correctness in Coq. Our certified kernel is written in 6500 lines of C and x86 assembly and runs on stock x86 multicore machines. To our knowledge, this is the first proof of functional correctness of a complete, general-purpose concurrent OS kernel with fine-grained locking.

1 Introduction

Operating System (OS) kernels and hypervisors form the backbone of safety-critical software systems in the world. Hence it is highly desirable to formally verify the correctness of these programs [53]. Recent efforts [33, 58, 34, 25, 23, 13, 5, 14] have shown that it is feasible to formally prove the functional correctness of simple general-purpose kernels, file systems, and device drivers, but none of these systems have addressed the important issues of concurrency [31, 7], which include not just user and I/O concurrency on a single CPU, but also multicore parallelism with fine-grained locking. This severely limits the applicability and power of today's formally verified system software.

What makes the verification of concurrent OS kernels so challenging? First, concurrent kernels allow interleaved execution of kernel/user modules across different abstraction layers; they contain many interdependent components that are difficult to untangle. Several researchers [55, 51] believe that the combination of concurrency and the kernels' functional complexity makes formal verification of functional correctness intractable, and even if it is possible, its cost would far exceed that of verifying a single-core sequential kernel.

Second, concurrent kernels need to support all three types of concurrency (user, I/O, or multicore) and make them work coherently with each other. User and I/O concurrency rely on thread yield/sleep/wakeup primitives or interrupts to switch control and support synchronization; these constructs are difficult to reason about since they transfer control from one thread to another. Multicore concurrency with fine-grained locking requires sophisticated spinlock implementations such as MCS locks [46], which are also hard to verify.

Third, concurrent kernels should also guarantee that each of their system calls eventually returns, but this depends on the progress of the concurrent primitives used in the kernels. Proving starvation-freedom [28] for concurrent objects only became possible recently [40]. Standard Mesa-style condition variables [35] do not guarantee starvation-freedom; this can be fixed by using a FIFO queue of condition variables, but the solution is not trivial and even the popular, most up-to-date OS textbook [7, Fig. 5.14] has gotten it wrong [6].

Fourth, given the high cost of building concurrent kernels, it is important that they can be quickly adapted to support new hardware platforms and applications [8, 45, 20]. One advantage of a certified kernel is the formal specification for all of its components. In theory, this allows us to add certified kernel plug-ins as long as they do not violate any existing invariants. In practice, however, if we are unable to encapsulate interference, even a small edit could incur huge verification overhead.

In this paper, we present a novel compositional approach that tackles all these challenges. We believe that, to control the complexity of concurrent kernels and to provide strong support for extensibility, we must first have a *compositional* specification that can untangle *all* the kernel interdependencies and encapsulate interference among different kernel objects. Because the very purpose of an OS kernel is to build layers of abstraction over bare machines, we insist on meticulously uncovering and specifying these layers, and then verifying each kernel module at its *proper* abstraction level.

The functional correctness of an OS kernel is often stated as a *refinement*. This is shown by building *forward simulation* [44] from the C/assembly implementation of a kernel (K) to its abstract functional specification (S). Of course, the ultimate goal of having a certified kernel is to reason about programs running on top of (or along with) the kernel. It is thus important to ensure that given any kernel extension or user program P, the combined code $K \bowtie P$ also refines $S \bowtie P$. If this fails to hold, the kernel is simply still incorrect since P can observe some difference between K and S. Gu et al. [23] advocated proving such a *contextual refinement* property, but they only considered the *sequential* contexts (i.e., P is sequential).

For concurrent kernels, proving the *contextual refinement* property becomes essential. In the sequential setting, the only way that the context code P can interfere with the kernel K is when K fails to encapsulate its private state; that is, P can modify some internal state of K without K's permission. In the concurrent setting, the *environment* context (ε) of a running kernel K could be other kernel threads or a copy of K running on another CPU. With shared-memory concurrency, the interference between ε and K are both necessary and often common; the sequential *atomic* specification S is now replaced by the notion of linearizability [29] plus a progress property such as starvation-freedom [28].

In fact, linearizability proofs often require event reordering that preserves the happens-before relation, so $K \bowtie \varepsilon$ may not even refine $S \bowtie \varepsilon$. Contextual refinement in the concurrent setting requires that for any ε, we can find a *semantically related* ε' such that $K \bowtie \varepsilon$ refines $S \bowtie \varepsilon'$. Several researchers [22, 42, 40] have shown that contextual refinement is precisely equivalent to the linearizability and progress requirements for implementing compositional concurrent objects [28, 29].

Our paper makes the following contributions:

- We present **CertiKOS**—a new extensible architecture for building certified concurrent OS kernels. CertiKOS uses contextual refinement over the "concurrent" *environment contexts* (ε) as the *unifying* formalism for composing different concurrent kernel/user objects at different abstraction levels. Each ε defines a specific instance on how other threads/CPUs/devices respond toward the events generated by the current running threads. Each abstraction layer, parameterized over a specific ε, is an assembly-level machine extended with a particular set of abstract objects (i.e., abstract states plus atomic primitives). CertiKOS successfully decomposes an otherwise prohibitive verification task into many simple and easily automatable ones.

- We show how the use of an environment context at each layer allows us to apply standard techniques for verifying sequential programs to verify concurrent programs. Indeed, most of our kernel programs are written in a variant of C (called ClightX) [23], verified at the source level, and compiled and linked together using CompCertX [23, 24]—a *thread-safe* version of the CompCert compiler [37, 38]. As far as we know, CertiKOS is the first architecture that can truly build certified concurrent kernels and transfer global properties proved for programs (at the kernel specification level) down to the concrete assembly machine level.

- We show how to impose temporal invariants over these environment contexts so we can verify the progress of various concurrent primitives. For example, to verify the starvation-freedom of ticket locks or MCS locks, we must assume that the multicore hardware (or the OS scheduler) always generates a *fair* interleaving, and those threads/CPUs which requested locks before the current running thread will eventually acquire and then release the lock. In a separate paper [24], we present the formal theory of environment contexts and show how these assumptions can be discharged when we compose different threads/CPUs to form a complete system.

- Using CertiKOS, we have successfully developed a fully certified concurrent OS kernel (called mC2) in the Coq proof assistant [2]. Our kernel supports both fine-grained locking and thread yield/sleep/wakeup primitives, and can run on stock x86 multicore machines. It can also double as a hypervisor and boot multiple instances of Linux in guest VMs running on different CPUs. Our certified hypervisor kernel consists of 6500 lines of C and x86 assembly. The entire proof effort for supporting concurrency took less than 2 person years. To our knowledge, this is the first proof of functional correctness of a complete, general-purpose concurrent OS kernel with fine-grained locking.

The rest of this paper is organized as follows. Section 2 gives an overview of our new CertiKOS architecture. Section 3 shows how we use environment contexts to turn concurrent layers into sequential ones. Section 4 presents the design and development of the mC2 kernel and how we verify various concurrent kernel objects. Section 5 presents an evaluation of CertiKOS. Sections 6-7 discuss related work and then conclude.

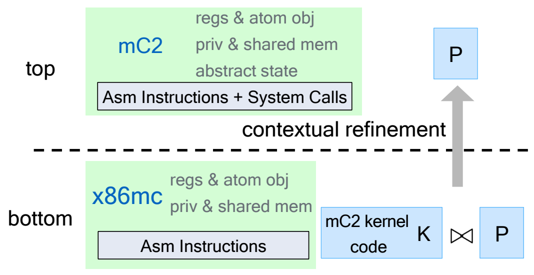


Figure 1: Certified OS kernels: what to prove?

2 Overview of Our Approach

The ultimate goal of research on building certified OS kernels is not just to verify the functional correctness of a particular kernel, but rather to find the best OS design and development methodologies that can be used to build provably reliable, secure, and efficient computer systems in a cost-effective way. We enumerate a few important dimensions of concerns and evaluation metrics which we have used so far to guide our work toward this goal:

- **Support for new kernel design.** Traditional OS kernels use the hardware-enforced "red line" to define a single system call API. A certified OS kernel opens up the design space significantly as it can support multiple certified kernel APIs at different abstraction levels. It is important to support kernel extensions [9, 20, 45] and novel ring-0 or guest-domain processes [30, 8] so we can experiment and find the best trade-offs.
- **Kernel performance.** Verification should not impose significant overhead on kernel performance. Of course, different kernel designs may imply different performance priorities. An L4-like microkernel [43] focuses on fast inter-process communication (IPC), while a Singularity-like kernel [30] emphasizes efficient support for type-safe ring-0 processes.
- **Verification of global properties.** A certified kernel is much less interesting if it cannot be used to prove global properties of the complete system built on top of the kernel. Such global properties include not only safety, liveness, and security properties of user-level processes and virtual machines, but also resource usage and availability properties (e.g., to counter denial-of-service attacks).
- **Quality of kernel specification.** A good kernel specification should capture precisely the *contextually observable* behaviors of the implementation [23]. It must support transferring global properties proved at a high abstraction level down to any lower abstraction level [16].
- **Cost of development and maintenance.** Compositionality is the key to minimize such cost. If the machine model is stable, verification of each kernel module should only need to be done once (to show that it *implements* its deep functional specification [23]). Global properties (e.g., information flow security) should be derived from the kernel deep specification alone [16].
- **Quality of formal proofs.** We use the term *certified kernels* rather than *verified kernels* to emphasize the importance of third-party machine-checkable proof certificates [53]. Hand-written paper proofs are error-prone [32]. Program verification without explicit machine-checkable proof objects has been subject to significant controversy [17].

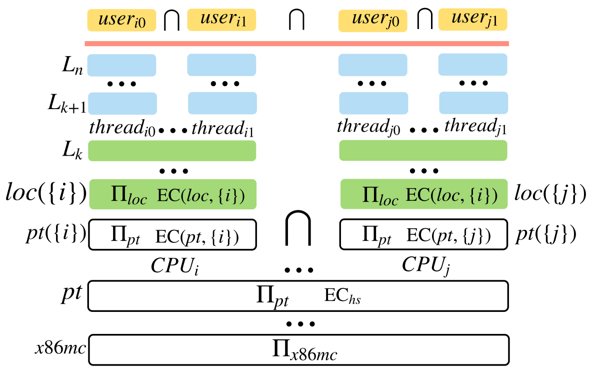


Figure 2: Contextual refinement between concurrent layers

Overview of CertiKOS Our new CertiKOS architecture aims to address all these concerns and also tackle the challenges described in Section 1. The CertiKOS architecture leverages the new certified programming methodologies developed by Gu et al. [23, 24] and applies them to support building certified concurrent OS kernels.

A *certified abstraction layer* consists of a language construct (L_1, M, L_2) and a mechanized proof object showing that the layer implementation M, built on top of the interface L_1 (the *underlay*), is a *contextual refinement* of the desirable interface L_2 above (the *overlay*). A *deep* specification (L_2) of a module (M) captures everything *contextually observable* about running the module over its underlay (L_1). Once we have certified M with a deep specification L_2, there is no need to ever look at M again, and any property about M can be proved using L_2 alone.

In Figure 1, we use x86mc to denote an assembly-level multicore machine. Suppose we load such a machine with the mC2 kernel K (in assembly) and user-level assembly code P, and we use $[[\cdot]]_{\text{x86mc}}$ to denote the whole-machine semantics for x86mc, then proving any global property of such a complete system amounts to reasoning about the semantic object $[[K \bowtie P]]_{\text{x86mc}}$, i.e., the set of observable behaviors from running $K \bowtie P$ on x86mc.

Reasoning at such a low level is difficult, so we formalize a new mC2 machine that extends the x86mc machine with the (deep) high-level specification of all system calls implemented by K. We use $[[\cdot]]_{\text{mC2}}$ to denote its whole-

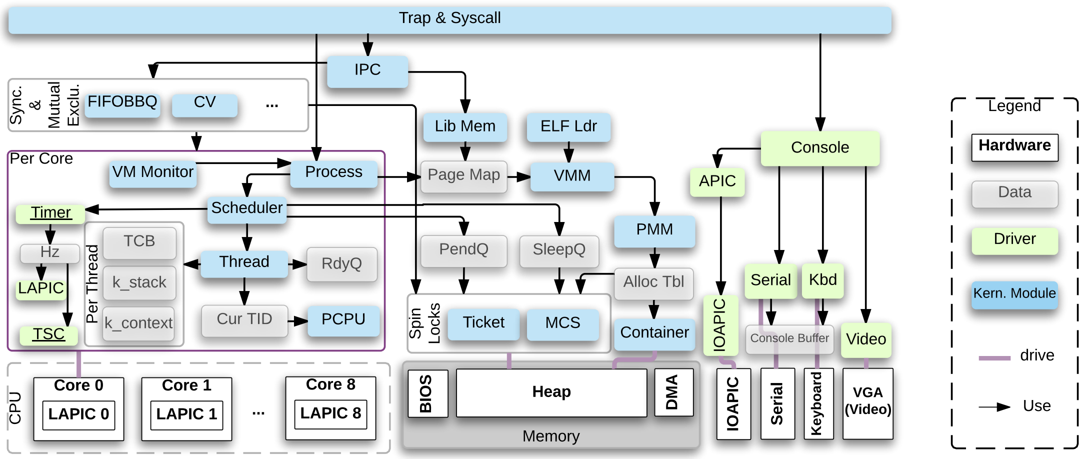


Figure 3: System architecture for the mC2 kernel

machine semantics. The contextual refinement property about the mC2 kernel can be stated as:

$$\forall P,\ [[K \ltimes P]]_{\text{x86mc}} \sqsubseteq [[P]]_{\text{mC2}}$$

Hence any global property proved about $[[P]]_{\text{mC2}}$ can be transferred to $[[K \ltimes P]]_{\text{x86mc}}$.

To support concurrency, for each layer interface L, we parameterize it with an *active* thread set A and then carefully define its set of valid *environment contexts*, denoted as $\text{EC}(L,A)$. Each environment context ε captures a specific instance—from a particular run—of the list of events that other threads or CPUs (i.e., those not in A) return when responding to the events generated by those in A. We can then define a new *thread-modular* machine $\Pi_{\text{L(A)}}(P,\varepsilon)$ that will operate like the usual assembly machine when P switches control to those threads in A, but will only obtain the list of events from the environment context ε when P switches control to those outside A. The semantics for a concurrent layer machine L is then:

$$[[P]]_{L(A)} = \{\ \Pi_{\text{L(A)}}(P,\varepsilon)\ \mid\ \varepsilon \in \text{EC}(L,A)\ \}$$

To support parallel layer composition, we carefully design $\text{EC}(L,A)$ so that the following property holds:

$$[[P]]_{L(A\cup B)}\ =\ [[P]]_{L(A)}\ \cap\ [[P]]_{L(B)}\ \ \textit{if}\ A\cap B=\varnothing$$

The formal details for $\text{EC}(L,A)$ and $[[\cdot]]_{L(A)}$ are presented in a separate paper [24]. Note that if A is a singleton, for each ε, $\Pi_{\text{L(A)}}$ behaves like a sequential machine.

With our new compositional layer semantics, we can take a multicore machine like x86mc and zoom into a specific active CPU i by creating a *logical* "single-core" machine layer for CPU i, and then apply techniques from Gu et al. [23] to build a collection of certified "sequential" (per-CPU) layers (see Figure 2). When we want to introduce kernel- or user-level threads, we can further zoom into a particular thread (e.g., $i0$) and create a corresponding logical machine layer. We can impose specific invariants over the environment contexts (i.e., the "rely" conditions) and use them to ensure that per-CPU or per-thread reasoning can be soundly composed (when their "rely" conditions are compatible with each other). After we have added all the kernel components and implemented all the system calls, we can combine these per-thread machines into a single concurrent machine.

Under CertiKOS, building a new certified concurrent kernel (or experimenting with a new design) is just a matter of composing a collection of certified concurrent layers, developed in a variant of C (called ClightX) or assembly. Gu et al. [23] have developed a certified compiler (CompCertX) that can compile certified ClightX layers into certified assembly layers. Since all concurrent primitives in CertiKOS are treated as CompCert-style external calls or built-ins, they cannot be reordered or optimized away by the compiler. Memory accesses over these external calls cannot be reordered either. Therefore, each concurrent ClightX module (running over a particular per-thread or per-CPU layer) is compiled by CompCertX as if it were a sequential program performing many external-call events. The correctness of CompCertX guarantees that the generated x86 assembly behaves the same as the source ClightX module. CompCertX can therefore serve as a *thread-safe* version of CompCert.

CertiKOS can thus enjoy the full programming power of both an ANSI C variant and an assembly language to certify any efficient routines required by low-level kernel programming. The layer mechanism allows us to certify most kernel components at higher abstraction levels, even though they all eventually get mapped (or compiled) down to an assembly machine.

Overview of the mC2 kernel Figure 3 shows the system architecture of mC2. The mC2 system was initially developed in the context of a large DARPA-funded research project. It is a concurrent OS kernel that can also double as a hypervisor. It runs on an Unmanned Ground Vehicle (UGV) with a multicore Intel Core i7 machine. On top of mC2, we run three Ubuntu Linux systems as guests (one each on the first three cores). Each virtual machine runs several RADL (The Robot Architecture Definition Language [39]) nodes that have fixed hardware capabilities such as access to GPS, radar, etc. The kernel also contains a few simple device drivers (e.g., interrupt controllers, serial and keyboard devices). More complex devices are either supported at the user level, or passed through (via IOMMU) to various guest Linux VMs. By running different RADL nodes in different VMs, mC2 provides strong isolation support so that even if attackers take control of one VM, they still cannot break into other VMs to compromise the overall mission of the UGV.

Within mC2, we have various shared objects such as spinlock modules (Ticket, MCS), sleep queues (SleepQ) for implementing queueing locks and condition variables, pending queues (PendQ) for waking up a thread on another CPU, container-based physical and virtual memory management modules (Container, PMM, VMM), a Lib Mem module for implementing shared-memory IPC, synchronization modules (FIFOBBQ, CV), and an IPC module. Within each core (the purple box), we have the per-CPU scheduler, the kernel-thread management module, the process management module, and the virtualization module (VM Monitor). Each kernel thread has its own thread-control block (TCB), context, and stack.

What have we proved? Using CertiKOS, we have successfully built a fully certified version of the mC2 kernel and proved its contextual refinement property with respect to a high-level deep specification for mC2. This important functional correctness property implies that all system calls and traps will strictly follow the high-level specification and always run *safely* and *terminate* eventually; and there will be no data race, no code injection attacks, no buffer overflows, no null pointer access, no integer overflow, etc.

More importantly, because for any program P, we have $[[K \bowtie P]]_{\text{x86mc}}$ refines $[[P]]_{\text{mC2}}$, we can also derive the important *behavior equivalence* property for P, that is, whatever behavior a user can deduce about P based on the high-level specification for the mC2 kernel K, the actual linked system $K \bowtie P$ running on the concrete x86mc machine would indeed behave exactly the same. All global properties proved at the system-call specification level can be transferred down to the lowest assembly machine.

Assumptions and limitations The mC2 kernel is obviously not as comprehensive as real-world kernels such as Linux. The main goal of this paper is to show that it is feasible to build certified concurrent kernels with fine-grained locking. We did not try to incorporate all the

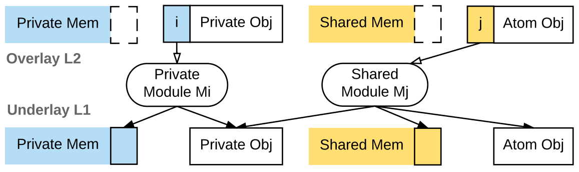


Figure 4: Defining concurrent abstraction layers

latest advances for multicore kernels into mC2.

Our assembly machine assumes strong sequential consistency for all atomic instructions. We believe our proof should remain valid for the x86 TSO model because (1) all our concurrent layers guarantee that non-atomic memory accesses are properly synchronized; and (2) the TSO order guarantees that all atomic synchronization operations are properly ordered. Nevertheless, more formalization work is needed to turn our proofs over sequential-consistent machines into those over the TSO machines [55].

Since our machine does not model TLB, any code for addressing TLB shootdown cannot be verified.

The mC2 kernel currently lacks a certified storage system. We plan to incorporate recent advances in building certified file systems [13, 5] into mC2 in the near future.

Our assembly machine only covers a small part of the full x86 instruction set, so our contextual correctness results only apply to programs in this subset. Additional instructions can be easily added if they have simple or no interaction with our kernel. Costanzo et al. [16, Sec. 6] shows how the fidelity of the CompCert-style x86 machine model would impact the formal correctness or security claims, and how such gap can be closed.

The CompCertX assembler for converting assembly into machine code is unverified. We assume correctness of the Coq proof checker and its code extraction mechanism.

The mC2 kernel also relies on a bootloader, a *PreInit* module (which initializes the CPUs and the devices), and an ELF loader. Their verification is left for future work.

3 Layer Design with Environment Context

In this section, we explain the general layer design principles and show how we use environment context to convert a concurrent layer into CPU-local layers.

Multicore hardware allows all the CPUs to access the same piece of memory simultaneously. In CertiKOS, we *logically* distinguish the *private memory* (i.e., private to a CPU or a thread) from the *shared memory* (i.e., shared by multiple CPUs or threads). The private memory does not need to be synchronized, whereas non-atomic shared memory accesses need to be protected by some synchronization mechanisms (e.g., locks), which are normally implemented using atomic hardware instructions (e.g., fetch-and-add). With proper protection, each shared memory operation can be viewed as if it were atomic.

Atomic object is an abstraction of well-synchronized shared memory, combined with operations that can be performed over that shared memory. It consists of a set of primitives, an initial state, and a *logical log* containing the entire history of the operations that were performed on the object during an execution. Each primitive invocation records a *single* corresponding event in the log. We require that these events contain enough information so we can derive the current state of each atomic object by replaying the entire log over the object's initial state.

Concurrent layer interface contains both *private objects* (e.g., i in Fig. 4) and *atomic objects* (e.g., j in Fig. 4), along with some invariants imposed on these objects. The verification of a concurrent kernel requires repeatedly building certified abstraction layers. The overlay interface L_2 is a new and more abstract interface, built on top of the underlay interface L_1, and implemented by module M_i or M_j (cf. Fig. 4). *Private objects* only access private memory and are built following techniques similar to those presented by Gu et al. [23]. *Atomic objects* are implemented by shared modules (e.g., M_j in Fig. 4) that may access existing atomic objects, private objects, and non-atomic shared memory.

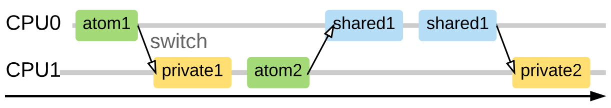
Every atomic primitive in the overlay generates exactly one event (this is why it is really atomic), while its implementation may trigger multiple events (by calling multiple atomic primitives in the underlay).

It is difficult to build certified abstraction layers directly on a multicore, nondeterministic hardware model. To construct an atomic object, we must reason about its implementation under all possible interleavings and prove that every access to shared memory is well synchronized.

In the rest of this section, we first present our x86 multicore machine model (Π_{x86mc}), and then show how we gradually refine this low-level model into a more abstract machine model (Π_{loc}) that is suitable for reasoning about concurrent code in a CPU-local fashion.

3.1 Multicore hardware model

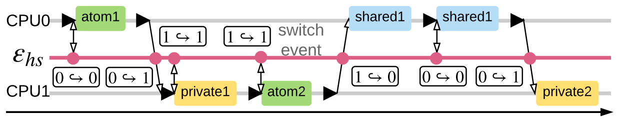
Our fine-grained multicore hardware model (Π_{x86mc}) allows arbitrary interleavings at the level of *assembly instructions*. At each step, the hardware *nondeterministically* chooses one CPU and executes the next assembly instruction on that CPU. Each assembly instruction is classified as *atomic*, *shared*, or *private*, depending on whether the instruction involves an atomic object call, a non-atomic shared memory access, or only a private object/memory access. One interleaving of an example program running on two CPUs is as follows:



Since only atomic operations generate events, this interleaving produces the logical log $[\texttt{0.atom}_1, \texttt{1.atom}_2]$.

3.2 Machine model with hardware scheduler

As a first step toward abstracting away the low-level details of the concurrent CPUs, we introduce a new machine model (Π_{hs}) configured with a *hardware scheduler* (ε_{hs}) that specifies a particular interleaving for an execution. This results in a deterministic machine model. To take a program from Π_{x86mc} and run it on top of Π_{hs}, we insert a *logical switch point* (denoted as "▶") before each assembly instruction. At each switch point, the machine first queries the hardware scheduler and gets the CPU ID that will execute next. All the *switch decisions* made by ε_{hs} are stored in the log as switch events. The previous example on Π_{x86mc} can be simulated by the following ε_{hs}:



The log recorded by this execution is as follows (a switch from CPU i to j is denoted as $i \hookrightarrow j$):

$$[0 \hookrightarrow 0, \texttt{0.atom}_1, 0 \hookrightarrow 1, 1 \hookrightarrow 1, 1 \hookrightarrow 1, \texttt{1.atom}_2, 1 \hookrightarrow 0, 0 \hookrightarrow 0, 0 \hookrightarrow 1]$$

The behavior of running a program P over this model with a hardware scheduler ε_{hs} is denoted as $\Pi_{\text{hs}}(P, \varepsilon_{hs})$, indicating that it is parametrized over all possible ε_{hs}. Let EC_{hs} represent the set of all possible hardware schedulers. Then we define the whole-machine semantics:

$$[[P]]_{hs} = \{\, \Pi_{\text{hs}}(P, \varepsilon_{hs}) \mid \varepsilon_{hs} \in \text{EC}_{hs} \,\}$$

Note this is a special case of the definition in Section 2 for the whole-machine semantics of a concurrent layer machine, where the active set is the set of all CPUs. To ensure correctness of this machine model with respect to the hardware machine model, we prove that Π_{x86mc} *contextually refines* the new model. Before we state the property, we first define *contextual refinement* formally.

Definition 1 (Contextual Refinement)**.** *We say that layer L_0 contextually refines layer L_1 (written as $\forall P, [[P]]_{L_0} \sqsubseteq [[P]]_{L_1}$), if and only if for any P that does not go wrong on Π_{L_1} under any configuration, we also have that (1) P does not go wrong on Π_{L_0} under any configuration; and (2) any observable behavior of P on Π_{L_0} under some configuration is also observed on Π_{L_1} under some (possibly different) configuration.*

Lemma 1 (Correctness of the hardware scheduler model)**.**

$$\forall P, [[P]]_{x86mc} \sqsubseteq [[P]]_{hs}$$

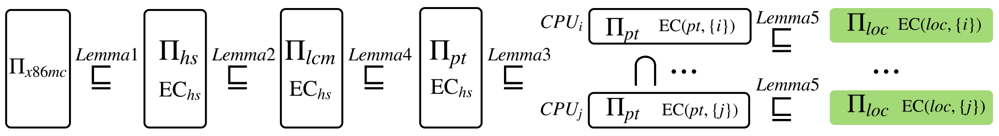


Figure 5: The contextual refinement chain from multicore hardware model Π_{x86mc} to CPU-local model Π_{loc}

3.3 Machine with local copy of shared memory

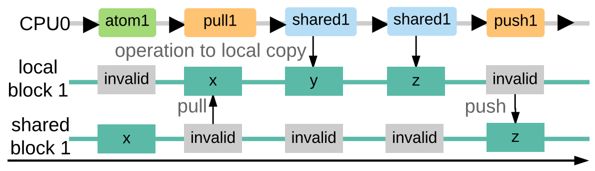
The above machine model does not restrict any access to the shared memory. We therefore abstract the machine model with hardware scheduler into a new model that enforces well-synchronized accesses to shared memory.

In addition to the global shared memory concurrently manipulated by all CPUs, each CPU on this new machine model (Π_{lcm}) also maintains a local copy of shared memory blocks along with a *valid bit*. The relation between a CPU's local copy and the global shared memory is maintained through two new *logical* primitives `pull` and `push`.

The `pull` operation over a particular CompCert-style memory block [37] updates a CPU's local copy of that block to be equal to the one in the shared memory, marking the local block as `valid` and the shared version as `invalid`. Conversely, the `push` operation updates the shared version to be equal to the local block, marking the shared version as `valid` and the local block as `invalid`.

If a program tries to pull an `invalid` shared memory block, push an `invalid` local block, or access an `invalid` local block, the program goes wrong. We make sure that every shared memory access is always performed on its `valid` local copy, thus systematically enforcing valid accesses to the shared memory. Note that all of these constructions are completely *logical*, and do not correspond to any physical protection mechanisms; thus they do not introduce any performance overhead.

The shared memory updates of the previous example can be simulated on Π_{lcm} as follows:



Data-race freedom Among each shared memory block and all of its local copies, only one can be `valid` at any single moment of machine execution. Therefore, for any program P with a potential *data race*, there exists a hardware scheduler such that P goes wrong on Π_{lcm}. By showing that a program P is safe (never goes wrong) on Π_{lcm} for *all possible* hardware schedulers, we guarantee that P is data-race free.

We have shown (in Coq) that Π_{lcm} is correct with respect to the previous machine model Π_{hs} with the EC_{hs}.

Lemma 2 (Correctness of the local copy model)**.**

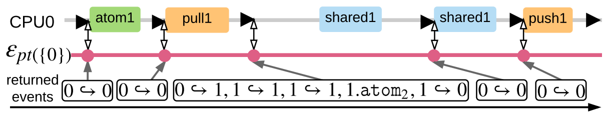
$$\forall P, [[P]]_{hs} \sqsubseteq [[P]]_{lcm}$$

3.4 Partial machine with environment context

Although Π_{lcm} provides a way to reason about shared memory operations, it still does not have much support for CPU-local reasoning. To achieve modular verification, the machine model should provide a way to reason about programs on each CPU locally by specifying expected behaviors of the context programs on other CPUs. The model should then provide a systematic way to link the proofs of different local components together to form a global claim about the whole system. To this purpose, we introduce a partial machine model Π_{pt} that can be used to reason about the programs running on a subset of CPUs, by parametrizing the model over the behaviors of an *environment context* (i.e., the rest of the CPUs).

We call a given local subset of CPUs the *active CPU set* (denoted as A). The partial machine model is configured with an active CPU set and it queries the environment context whenever it reaches a switch point that attempts to switch to a CPU outside the active set.

The set of **environment contexts** for A in this machine model is denoted as $\text{EC}(pt, A)$. Each environment context $\varepsilon_{pt(A)} \in \text{EC}(pt, A)$ is a *response function*, which takes the current log and returns a list of events from the context programs (i.e., those outside of A). The response function simulates the observable behavior of the context CPUs and imposes some invariants over the context. The hardware scheduler is also a part of the environment context, i.e., the events returned by the response function include switch events. The execution of CPU 0 in the previous example can be simulated with a $\varepsilon_{pt(\{0\})}$ function:



For example, at the 3rd switch point, $\varepsilon_{pt(\{0\})}$ returns the event list $[0 \hookrightarrow 1, 1 \hookrightarrow 1, 1 \hookrightarrow 1, 1.\texttt{atom}_2, 1 \hookrightarrow 0]$.

Composition of partial machine models Suppose we have verified that two programs, separately running with two *disjoint* active CPU sets A and B, produce event lists satisfying invariants $\texttt{INV}_A$ and $\texttt{INV}_B$, respectively. If $\texttt{INV}_A$ is consistent with the environment-context invariant of

B, and $\texttt{INV}_B$ is consistent with the environment-context invariant of A, then we can compose the two separate programs into a single program with active set $A \cup B$. This combined program is guaranteed to produce event lists satisfying the combined invariant $\texttt{INV}_A \wedge \texttt{INV}_B$. Using the whole-machine semantics from Section 2, we express this composition as a contextual refinement.

Lemma 3 (Composition of partial machine models)**.**

$$\forall P, [[P]]_{pt(A \cup B)} \sqsubseteq [[P]]_{pt(A)} \cap [[P]]_{pt(B)} \quad \textit{if } A \cap B = \varnothing$$

After composing the programs on all CPUs, the context CPU set becomes empty and the composed invariant holds on the whole machine. Since there is no context CPU, the environment context is reduced to the *hardware scheduler*, which only generates the switch events. In other words, letting C be the entire CPU set, we have that $\mathrm{EC}(pt, C) = \mathrm{EC}_{hs}$. By showing that this *composed machine* with the entire CPU set C is refined by Π_{lcm}, the proofs can be propagated down to the multicore hardware model.

Lemma 4 (Correctness of the composed total machine)**.**

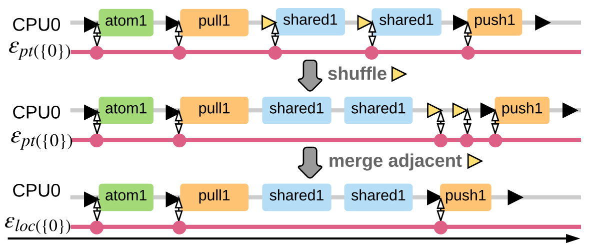
$$\forall P, [[P]]_{lcm} \sqsubseteq [[P]]_{pt(C)}$$

3.5 CPU-local machine model

If we focus on a single active CPU i, the partial machine model is like a *local* machine with an environment context representing all other CPUs. However, in this model there is a switch point before each instruction, so program verification still needs to handle many unnecessary interleavings (e.g., those between private operations). In this subsection, we introduce a CPU-local machine model (denoted as Π_{loc}) for a CPU i, in which switch points only appear before atomic or `push/pull` operations. The switch points before shared or private operations are removed via two steps: *shuffling* and *merging*.

Shuffling switch points In Π_{loc}, we introduce a *log cache* — for the switch points before shared and private operations, the query results from the environment context are stored in a temporary log cache. The cached events are applied to the logical log just before the next atomic or `push/pull` operation. Thus, when we perform shared or private operations, the observations of the environment context are delayed until the next atomic or `push/pull` operation. This is possible because a shared operation can only be performed when the current local copy of shared memory is valid, meaning that no other context program can interfere with the operation.

Merging switch points Once the switch points are shuffled properly, we merge all the adjacent switch points together. When we merge switch points, we also need to merge the switch events generated by the environment context. For example, the change of switch points for the previous example on CPU-local machine is as follows:



Lemma 5 (Correctness of CPU-local machine model)**.**

$$\forall P, [[P]]_{pt(\{i\})} \sqsubseteq [[P]]_{loc(\{i\})}$$

Finally, we obtain the refinement relation from the multicore hardware model to the CPU-local machine model by composing all of the refinement relations together (cf. Fig. 5). We introduce and verify the mC2 kernel on top of the CPU-local machine model Π_{loc}. The refinement proof guarantees that the proved properties can be propagated down to the multicore hardware model Π_{x86mc}.

All our proofs (including every step in Fig. 5 and Fig. 2) are implemented, composed, and machine-checked in Coq. Each refinement step is implemented as a CompCert-style upward-forward simulation from one layer machine to another. Each machine contains the usual (CPU-local) abstract state, a logical global log (for shared state), and an environment context. The simulation relation is defined over these two machine states, and matches well the informal intuitions given in this and next sections.

4 Certifying the mC2 Kernel

Contextual refinement provides an elegant formalism for decomposing the verification of a complex kernel into a large number of small tractable tasks: we define a series of logical abstraction layers, which serve as increasingly higher-level specifications for an increasing portion of the kernel code. We design these abstraction layers in a way such that complex interdependent kernel components are untangled and converted into a well-organized kernel-object stack with clean specification (cf. Fig. 2).

In the mC2 kernel, the pre-initialization module is the bottom layer that connects to the *CPU-local machine model* Π_{loc}, instantiated with a particular *active CPU* (cf. Sec. 3.5). The trap handler contains the top layer that provides system call interfaces and serves as a specification of the whole kernel, instantiated with a particular active thread running on that active CPU. Our main theorem states that any global properties proved at the topmost abstraction layer can be transferred down to the lowest hardware machine. In this section, we explain selected components in more details.

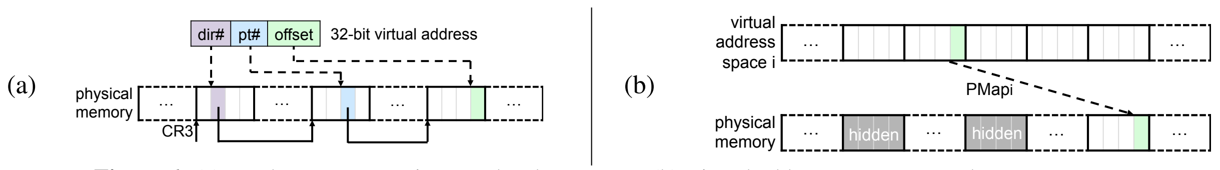
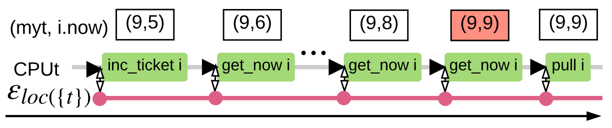


Figure 6: (a) Hardware MMU using two-level page map; (b) Virtual address space *i* set up by page map *i*

Each CPU-local pre-initialization machine defines the x86 hardware behaviors including page table walk upon memory load (when paging is turned on), saving and restoring the trap frame in the case of interrupts and exceptions (e.g., page fault), and the data exchange between devices and memory. The hardware memory management unit (MMU) is modeled in a way that mirrors the paging hardware (cf. Fig. 6a). When paging is enabled, memory accesses made by both the kernel and the user programs are translated using the page map pointed to by `CR3`. When a page fault occurs, the fault information is stored in `CR2`, the CPU mode is switched from user mode to kernel mode, and the page fault handler is triggered.

The spinlock module provides fine-grained lock objects as the base of synchronization mechanisms.

Ticket Lock depends on an *atomic ticket object*, which consists of two fields: `ticket` and `now`. Figure 7 shows one implementation of a ticket lock. Here, `L` is declared as an array of ticket locks; each shared data object can be protected with one lock in the array, identified using a specific lock index (`i`). The atomic increment to the ticket is achieved through the atomic `fetch-and-increment` (FAI) operation (implemented using the `xaddl` instruction with the `lock` prefix in x86). As described in Section 3.5, the *switch points* at this abstraction level have been shuffled and merged so that there is exactly one switch point before each atomic operation. Thus, the lock implementations generate a list of events; for example, when CPU *t* acquires the lock *i* (stored in `L[i]`), it continuously generates the event "`t.get_now i`" (line 10) until the latest `now` is increased to the ticket value returned by the event "`t.inc_ticket i`" (line 9), and then followed by the event "`t.pull i`" (line 11):



The event list is as below:

[▶,`t.inc_ticket i`, ▶,`t.get_now i`,···, ▶,`t.get_now i`]

Verifying the linearizability and starvation-freedom of the ticket lock object is equivalent to proving that under a *fair* hardware scheduler ε_{hs}, the ticket lock implementation is a *termination-sensitive* contextual refinement of its atomic specification [42, 40]. There are two main proof

```
1 typedef struct {
2   volatile uint ticket;
3   volatile uint now;
4 } ticket_lock;
5 ticket_lock L[NUM_LOCK];
6
7 void acq_lock (uint i) {
8   uint t;
9   t=▶FAI(&L[i].ticket);
10  while(▶L[i].now!=t){}
11  ▶pull (i);
12 }
13 void rel_lock (uint i) {
14   ▶push (i);
15   ▶L[i].now ++;
16 }
```

Figure 7: Pseudocode of the ticket lock implementation

obligations: (1) the lock guarantees *mutual exclusion*, and (2) the `acq_lock` operation eventually succeeds.

Mutual exclusion is straightforward for a ticket lock. At any time, only the thread whose ticket is equal to the current serving ticket (i.e., `now`) can hold the lock. Furthermore, each thread's ticket is unique as the `fetch-and-increment` operation is atomic (line 9). Thanks to this *mutual exclusion* property, it is safe to *pull* the shared memory associated with the lock *i* to the local copy at line 11. Before releasing the lock, the local copy is *pushed* back to the shared memory at line 14.

To prove that `acq_lock` eventually succeeds, from the fairness of ε_{hs}, we assume that between any two consecutive events from the same thread, there are at most *m* events generated by other threads (for some *m*). We also impose the following invariants on the environment:

Invariant 1 (Invariants for ticket lock). *An environment context that holds the lock i (1) never acquires lock i again before releasing it; and (2) always releases lock i within k steps (for some k).*

Lemma 6 (Starvation-freedom of ticket lock). *Acquiring ticket-lock in the mC2 kernel eventually succeeds.*

Proof. The full proofs are mechanized in Coq; here we highlight the main ideas. Let *n* be the maximum number of the total threads. Then (1) there are at most *n* threads waiting before the current one; (2) the thread holding the lock releases the lock within *k* steps, which generates at most *k* events; and (3) the environment context generates at most *m* events between each step of the lock holder. Hence there are at most $n \times m \times k$ events generated by the *context* of the threads waiting before the current one. Since the current thread belongs to this "context" and each read to the `now` field generates one `get_now` event, there are at most $n \times m \times k$ loop iterations at line 10 in Fig. 7. Thus, acquiring lock always succeeds. □

After we abstract the lock implementation into an atomic specification, each acquire-lock call in the higher layers only generates a single event "`t.acq_lock i`." We can compose such per-CPU specification with those of its environment CPUs as long as they all follow Invariant 1.

MCS Lock is known to have better scalability than ticket lock over machines with a larger number of CPUs. In mC2, we have also implemented a version of MCS locks [46]. The starvation-freedom proof is similar to that of the ticket lock. The difference is that the MCS lock-release operation waits in a loop until the next waiting thread (if it exists) has added itself to a linked list, so we need similar proofs for both acquire and release.

Physical memory management introduces the page allocation table `AT` (with `nps` denoting the maximum physical page number). Since `AT` is shared among different CPUs, we associate it with a lock `lock_AT`. The page allocator is then refined into an atomic object where the implementation for each of its methods (e.g., `palloc` in Fig. 8) is proved to satisfy an atomic interface, with the proof that lock utilization for `lock_AT` satisfies Inv. 1. Once the atomic allocator is introduced, lock acquire and release for `lock_AT` are *not allowed to be invoked* at higher layers. Thus, in this layered approach, it is not possible that a thread holding a lock defined at a lower layer tries to acquire another lock introduced at a higher layer, i.e., the order that a thread acquires different locks is guided by the layer order that the locks are introduced. This implicit order of lock acquisitions prevents *deadlocks* in mC2.

Another function of the physical memory management is to dynamically track and bound the memory usage of each thread. A *container* object is used to record information for each thread (array `cn` in Fig. 8); one piece of information tracked is the thread's *quota*. Inspired by the notions of containers and quotas in HiStar [59], a thread in mC2 is spawned with some quota specifying the maximum number of pages that the thread will ever be allowed to allocate. As can be seen in Fig. 8, `palloc` returns an error code if the requesting thread has no remaining quota (lines 2 and 3), and the quota is decremented when a page is successfully allocated (line 13). Quota enforcement allows the kernel to prevent a denial-of-service attack, where one thread repeatedly allocates pages and uses up all available memory (thus denying other threads from allocating pages). From a security standpoint [16], it also prevents the undesirable information channel between different threads that occurs due to such an attack.

Virtual memory management provides consecutive virtual address spaces on top of physical memory management (see Fig. 6b), We prove that the primitives manipulating page maps are correct, and the *initialization procedure* sets up the two-level page maps properly in terms of the hardware address translation.

```
1  int palloc (uint tid) {
2    if (cn[tid].quota < 1)
3      return ERROR;
4    ▶acq_lock (lock_AT);
5    uint i=0,fp=nps;
6    while(fp==nps&&i<nps){
7      if (!AT[i].free)
8        fp = i;
9      i++; }
10   if (fp != nps) {
11     AT[i].free = 0;
12     AT[i].ref = 1;
13     cn[tid].quota --;
14   }
15   else fp = ERROR;
16   ▶rel_lock (lock_AT);
17   return fp;
18 }
```

Figure 8: Pseudocode of `palloc`

Invariant 2. *(1) paging is enabled only after all the page maps are initialized; (2) pages that store kernel-specific data must have the kernel-only permission in all page maps; (3) the kernel page map is an identity map; and (4) non-shared parts of user processes' memory are isolated.*

By Inv. 2, we show that it is safe to run both the kernel and user programs in the virtual address space when paging is enabled. In this way, memory accesses at higher layers operate on the basis of the high-level, abstract descriptions of address spaces rather than concrete page directories and page tables stored in the memory itself.

Shared memory management provides a protocol to share physical pages among different user processes. A physical page can be mapped into multiple processes' page maps. For each page, we maintain a *logical owner set*. For example, a user process k_1 can share its private physical page i to another process k_2 and the logical owner set of page i is changed from $\{k_1\}$ to $\{k_1, k_2\}$. A shared page can only be freed when its owner set is a *singleton*.

The shared queue library abstracts the queues implemented as *doubly-linked lists* into *abstract queue states* (i.e., Coq lists). The local *enqueue* and *dequeue* operations are specified over the abstract lists. As usual, we associate each shared queue with a lock. The atomic interfaces for shared queue operations are represented by queue events "`t.enQ i e`" and "`t.deQ i`", which can be replayed to construct the shared queue. For instance, starting from an empty initial queue, if the current log of the i-th shared queue is $[\blacktriangleright, \texttt{t}_0\texttt{.enQ i 2}, \blacktriangleright, \texttt{t}_0\texttt{.deQ i}]$, and the event lists generated by the *environment context* at two switch points are $[\texttt{t}_1\texttt{.enQ i 3}]$ and $[\texttt{t}_1\texttt{.enQ i 5}]$, respectively, then the complete log for the queue i is:

$$[\texttt{t}_1\texttt{.enQ i } 3, \texttt{t}_0\texttt{.enQ i } 2, \texttt{t}_1\texttt{.enQ i } 5, \texttt{t}_0\texttt{.deQ i}]$$

By replaying the log, the shared queue state becomes $[2,5]$, and the last atomic dequeue operation returns 3.

Thread management introduces the thread control block and manages the resources of dynamically spawned threads (e.g., quotas) and their meta-data (e.g., children, thread state). For each thread, one page (4KB) is allocated for its *kernel stack*. We use an external tool [12] to show

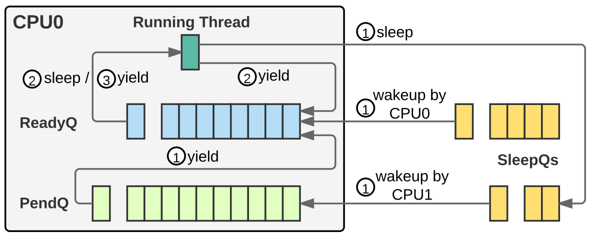


Figure 9: Scheduling routines `yield`, `sleep`, and `wakeup`

that the stack usage of our compiled kernel is less than 4KB, so stack overflows cannot occur inside the kernel.

One interesting aspect of the thread module is the context switch function. This assembly function saves the register set of the current thread and restores the register set from the kernel context of another thread on the same CPU. Since the instruction pointer register (`EIP`) and stack pointer register (`ESP`) are saved and restored in this procedure, this kernel context switch function is verified at the assembly level, and linked with other code that is verified at the C level and then compiled by CompCertX.

The thread scheduling is done by three primitives: `yield`, `sleep`, and `wakeup`. They are implemented using the shared queue library (cf. Fig. 9). Each CPU has a *private ready queue* ReadyQ and a *shared pending queue* PendQ. The context CPUs can insert threads to the current CPU's pending queue. The mC2 kernel also provides a set of shared *sleeping queues* SleepQs. As shown in Fig. 9, `yield` moves a thread from the pending queue to the ready queue and then switches to the next ready thread. The `sleep` primitive simply adds the running thread to a sleeping queue and runs the next ready thread. The `wakeup` primitive contains two cases. If the thread to be woken up belongs to the current CPU, then the primitive adds the thread to its ready queue. Otherwise, `wakeup` adds the thread to the pending queue of the CPU it belongs to. Except for the ready queue, all the other thread queue operations are protected by *fine-grained* locks.

Thread-local machine models can be built based on the thread management layers. The first step is to extend the environment context with a *software scheduler* (i.e., abstracting the concrete scheduling procedure), resulting in a new environment context ε_{ss}. The scheduling primitives generate the `yield` and `sleep` events and ε_{ss} responds with the next thread ID to execute. One invariant we impose on ε_{ss} is that a sleeping thread can be rescheduled only after a `wakeup` event is generated. The second step is to introduce the *active thread set* to represent the *active* threads on the *active CPU*, and extend the ε_{ss} with the *context threads*, i.e., the rest of the threads running on the active CPU. The composition structure is similar to the one of Lemma 3. In this way, higher layers can be built upon a thread-local machine model with a single active thread on the active CPU (cf. Fig. 2).

```
1  struct fifobbq {
2    Queue insrtQ, rmvQ;
3    int n_rmv, n_insrt;
4    int front, next;
5    int T[MAX]; lock l;
6  } q;
7
8  void remove(){
9    uint cv, pos, t;
10   ▶acq_lock (q.l);
11   pos = q.n_rmv ++;
12   cv = my_cv ();
13   ▶enQ (q.rmvQ, cv);
14   while(q.front < pos ||
15         q.front == q.next)
16     ▶wait (cv, q.l);
17
18   t = q.T[q.front % MAX]
19   q.front ++;
20
21   cv=▶peekQ (q.insrtQ);
22   if (cv != NULL)
23       ▶signal (cv);
24   ▶deQ (q.rmvQ);
25   cv = ▶peekQ (q.rmvQ);
26   if (cv != NULL)
27       ▶signal (cv);
28   ▶rel_lock (q.l);
29   return t;
30 }
```

Figure 10: Pseudocode of the remove method for FIFOBBQ

Starvation-free condition variable A *condition variable* (CV) is a synchronization object that enables a thread to wait for a change to be made to a shared state (protected by a lock). Standard Mesa-style CVs [35] do not guarantee starvation-freedom: a thread waiting on a CV may not be signaled within a bounded number of execution steps. We have implemented a starvation-free version of CV using condition queues as shown by Anderson and Dahlin [7, Fig. 5.14]. However, we have found a bug in the FIFOBBQ implementation shown in that textbook: in some cases, their system can get stuck by allowing all the signaling and waiting threads to be asleep simultaneously, or the system can arrive at a dead end where the threads on the remove queue (rmvQ) can no longer be woken up. We fixed this issue by postponing the removal of the CV of a waiting thread from the queue, until the waiting thread finishes its work (cf. Fig. 10); the remover is now responsible for removing itself from the rmvQ (line 24) and waking up the next element in the rmvQ (line 27). Here, `peekQ` reads the head item of a queue; and `my_cv` returns the CV assigned to the current running thread.

5 Evaluation

Proof effort and the cost of change We take the certified sequential mCertiKOS kernel [23], and extend the kernel with various features such as dynamic memory management, container support for controlling resource consumption, Intel hardware virtualization support, shared memory IPC, single-copy synchronous IPC, ticket and MCS locks, new schedulers, condition variables, etc. Some of these features were initially added in the sequential setting but later ported to the concurrent setting. During this development process, many of our certified layers (including their implementation, their functional specification, and the layer refinement proofs) have undergone many rounds of modifications and extensions.

CertiKOS makes such evolution process much easier. For example, all certified layers in the sequential kernel can be directly ported to the concurrent setting if they do not use any synchronization. We have also merged the work by Chen et al. [14] on the interruptible kernel with device drivers using our multicore model.

Overall, our certified mC2 kernel consists of 6500 lines of C and x86 assembly. We have also developed a general linking theorem for composing multiple threads running on the same CPU, and another theorem for combining programs running on different CPUs. Our team completed the verification of the new concurrency framework and features in about 2 person years.

Regarding specification, there are 943 lines of code used to specify the lowest layer axiomatizing the hardware machine model, and 450 lines of code for the specification of the abstract system call interfaces. These are in our trusted computing base. We keep these specifications small to limit the room for errors and ease the review process. Outside the trusted computing base, there are 5249 lines of additional specifications for the various kernel functions, and about 40K lines of code used to define auxiliary definitions, lemmas, theorems, and invariants. Additionally, there are 50K lines of Coq proof scripts for proving the newly-added concurrency features. At least one third of these auxiliary definitions and proof scripts are redundant and semi-automatically generated, which makes our proof a little verbose. For example, many invariant proofs get duplicated across the layers whenever there is a minor change to the entire set of invariants. We are currently working on a new layer calculus to minimize redundant definitions and proofs.

Bugs found Other than the FIFOBBQ bug, we have also found a few other bugs during verification. Our initial ticket-lock implementation contains a particularly subtle bug: the spinning loop body (line 10 in Fig. 7) was implemented as `while( ▶L[i].now<t){}`. This passed all our tests, but during the verification, we found that it did not satisfy the atomic specification since the ticket field might overflow. For example, if `L[i].ticket` is $(2^{32}-1)$, `acq_lock` will cause an overflow (line 9 in Fig. 7) and the returned ticket `t` equals 0. In this case, `L[i].now` is not less than `t` and `acq_lock` returns immediately, which violates the order implied by the ticket. We fixed this bug by changing the loop body to "`while( ▶L[i].now!=t){}`"; we completed the proof by showing that the maximum number of concurrent threads is far below 2^{32}.

Performance evaluation Although the performance is not the main emphasis of this paper, we have run a number of micro and macro benchmarks to measure the speedup and overhead of mC2 and to compare mC2 to existing systems such as KVM and seL4. All experiments have been performed on an Intel Core i7-2600S (2.8GHz, 4 cores) with 8 MB L3 cache, 16 GB memory, and a 120 GB Intel 520 SSD. Since the power control code has not been verified, we disabled the turbo boost and power management features of the hardware during experiments.

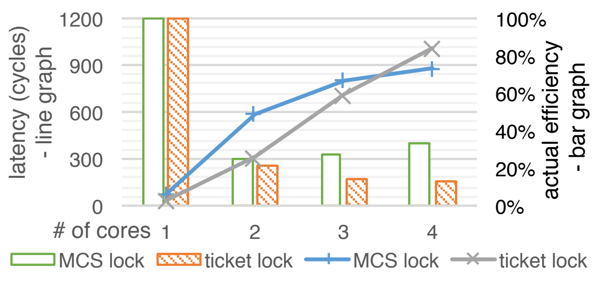


Figure 11: The comparison between actual efficiency of ticket lock and MCS lock implementations in mC2

Concurrency overhead The run-time overhead introduced by concurrency in mC2 mainly comes from *the latency of spinlocks* and *the contention of the shared data*.

The mC2 kernel provides two kinds of spinlocks: ticket lock and MCS lock. They have the same interface and thus are interchangeable. In order to measure their performance, we put an empty critical section (payload) under the protection of a single lock. The latency is measured by taking a sample of 10,000 consecutive lock acquires and releases (transactions) on each round.

Figure 11 shows the results of our latency measurement. In the single core case, ticket locks impose 34 cycles of overhead, while MCS locks impose 74 cycles (line chart). As the number of cores grows, the latency increases rapidly. However, note that all transactions are protected by the same lock. Thus, it is expected that the slowdown should be proportional to the number of cores. In order to show the actual efficiency of the lock implementations, we normalize the latency against the baseline (single core) multiplied by the number of cores ($\frac{n*t_1}{t_n}$). As can be seen from the bar chart, efficiency remains about the same for MCS lock, but decreases for ticket lock.

Now that we have compared MCS lock with ticket lock, we present the remaining evaluations in this section using only the ticket lock implementation of mC2.

To reduce contention, all shared objects in mC2 are carefully designed and pre-allocated with a fine-grained lock. We design a benchmark with server/client pairs to evaluate the speedup of the system as more cores are introduced. We run a pair of server/client processes on each core, and we measure the total throughput (i.e., the number of transactions that servers make in each millisecond) across all available cores. A server's transaction consists of first performing an IPC receive from a channel i, then executing a payload (certain number of '`nop`' instructions), and finally sending a message to channel $i+1$. Correspondingly, a client executes a constant payload of 500 cycles, sends an IPC message to channel i, and then

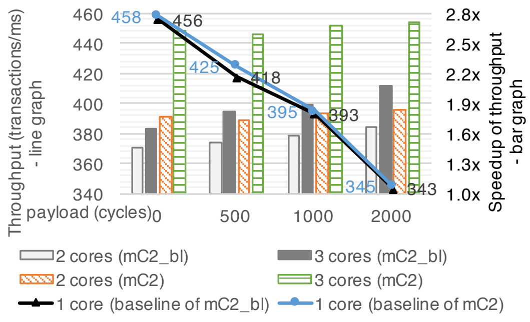


Figure 12: Speedup of throughput of mC2 vs. mC2-bl in a client/server benchmark under various server payloads (0-2,000)

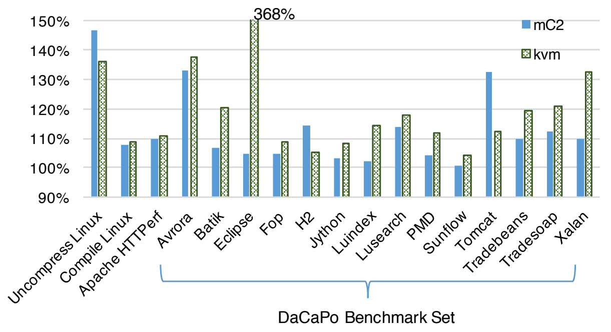


Figure 13: Normalized performance for macro benchmarks running over Linux on KVM vs. Linux on mC2; the baseline is Linux on bare metal; a smaller ratio is better

receives its server's message through channel $i+1$. When the client has to wait for a reply from the server, the control is switched to a special system process which then immediately yields back to the server process.

Figure 12 shows this server/client benchmark, comparing mC2 against a big-kernel-lock version of mC2 (mC2-bl). We insert a pair of lock acquire and release at the top-most layer by hand, and replace all fine-grained locks with an empty function. This does not introduce bias because the speedup is normalized against its own baseline (single core throughput) for each kernel version separately. From the figure, we can see that the speedup rate for big-kernel-lock is about 1.45x ~ 1.66x with 2 cores and 1.64x ~ 2.07x with 3 cores. On the other hand, the fine-grained locks of mC2 yield better speedup as the number of cores increases (roughly 1.77x ~ 1.84x and 2.62x ~ 2.71x with 2 and 3 cores, respectively). Note that the server/client pairs are distributed into different CPUs, and there is no cross core communication; therefore, one might expect perfect scaling as the number of cores increases. We did not quite achieve this because each core must execute some system processes which run at constant rates and consume CPU resources, and we did not align kernel data structures against cache-line size.

IPC Performance We measure the latency of IPC send/recv in mC2 against various message sizes, and compare the result with seL4's IPC implementation.

A comparison of the performance of seL4 and mC2 is not straightforward since the verified mC2 kernel runs on a multicore x86 platform, while the verified seL4 kernel runs on ARMv6 and ARMv7 hardware and only supports single-core. Thus, we use an unverified, single-core version of seL4 for comparison. Moreover, the synchronized IPC API in seL4 (`Call/ReplyWait`) has a different semantics from mC2's send/recv: it uses a round-trip message passing protocol (with a one-off reply channel created on the fly) while trapping into the kernel twice, and it does not use any standard sleep or wakeup procedures. To have a meaningful comparison with respect to the efficiency of implementing system calls, we compare $(send + recv) \times 2$ of mC2 with $(Call + ReplyWait) + Null \times 2$ of seL4, where *Null* is the latency of a null system call in seL4.

We measure seL4's performance using seL4's IPC benchmark sel4bench-manifest [3] with processes in different address spaces and with identical scheduler priorities, both in *slowpath* and *fastpath* configurations. We consulted the seL4 team [27] and used 158 cycles as the cost of each null system call (*Null*) in seL4. To measure mC2's performance, we simply replace seL4's *Call* and *ReplyWait* system calls with mC2's synchronous *send* and *receive* calls. We found that, when the buffer size is zero, mC2 takes about 3800 cycles to perform a round trip IPC, while seL4's fastpath IPC takes roughly 1200 cycles, and seL4's slowpath IPC takes 1800 cycles. When the message size is larger than 2 words, the fastpath IPC of seL4 falls back to the slowpath; in the 10-words IPC case, mC2's round trip IPC takes 3820 cycles, while seL4 takes 1830 cycles. Note that seL4 follows the microkernel design philosophy, and thus its IPC performance is critical. IPC implementations in seL4 are highly optimized and heavily tailored to specific hardware platforms.

Hypervisor Performance To evaluate mC2 as a hypervisor, we measured the performance of some macro benchmarks on Ubuntu 12.04.2 LTS running as a guest. We ran the benchmarks on Linux as guest in both KVM and mC2, as well as on the bare metal. The guest Ubuntu is installed on an internal SSD drive. KVM and mC2 are installed on a USB stick. We use the standard 4KB pages in every setting — huge pages are not used.

Figure 13 contains a compilation of standard macro benchmarks: unpacking of the Linux 4.0-rc4 kernel, compilation of the Linux 4.0-rc4 kernel, Apache HTTPerf [47] (running on loopback), and DaCaPo Benchmark 9.12 [11]. We normalize the running times of the benchmarks using the bare metal performance as a baseline (100%). The overhead of mC2 is moderate and comparable to KVM. In some cases, mC2 performs better than KVM; we suspect this is because KVM has a Linux host and thus has a larger cache footprint. For benchmarks with a large number of file operations, such as Uncompress Linux source and Tomcat, mC2 performs worse. This is because mC2 expose the raw disk interface to the guest via VirtIO [52] (instead of doing the pass-through), and its disk driver does not provide good buffering support.

6 Related Work

Dijkstra [18, 19] proposed to "realize" a complex program by decomposing it into a hierarchy of linearly ordered abstract machines. Based on this idea, the PSOS team at SRI [48] developed the Hierarchical Development Methodology (HDM) and applied it to design and specify an OS using 20 hierarchically organized modules. HDM was later also used for the KSOS system [50]. Gu et al. [23] developed new languages and tools for building certified abstraction layers with *deep* specifications, and showed how to apply the layered methodology to construct fully certified (sequential) OS kernels in Coq.

Costanzo et al. [16] showed how to prove sophisticated global properties (e.g., information-flow security) over a deep specification of a certified OS kernel and then transfer these properties from the specification level to its correct assembly-level implementation. Chen et al. [14] extended the layer methodology to build certified kernels and device drivers running on multiple *logical* CPUs. They treat the driver stack for each device as if it were running on a logical CPU dedicated to that device. Logical CPUs do not share any memory, and are all eventually mapped onto a single physical CPU. None of these systems, however, can support shared-memory concurrency with fine-grained locking.

The seL4 team [33, 34] was the first to verify the functional correctness and security properties of a high-performance L4-family microkernel. The seL4 microkernel, however, does not support multicore concurrency with fine-grained locking. Peters et al. [51] and von Tessin [55] argued that for an seL4-like microkernel, concurrent data accesses across multiple CPUs can be reduced to a minimum, so a single *big kernel lock (BKL)* might be good enough for achieving good performance on multicore machines. von Tessin [55] further showed how to convert the single-core seL4 proofs into proofs for a BKL-based clustered multikernel.

The Verisoft team [49, 36, 4] applied the VCC framework [15] to formally verify Hyper-V, which is a widely deployed multiprocessor hypervisor by Microsoft consisting of 100 kLOC of concurrent C code and 5 kLOC of assembly. However, only 20% of the code is verified [15]; it is also only verified for function contracts and type invariants, not the full functional correctness property. There is a large body of other work [10, 58, 25, 13, 26, 56, 5, 54] showing how to build verified OS kernels, hypervisors, file systems, device drivers, and distributed systems, but they do not address the issues on concurrency.

Xu et al. [57] developed a new verification framework by combining rely-guarantee-based simulation [41] with Feng et al.'s program logic for reasoning about interrupts [21]. They have successfully verified key modules in the μC/OS-II kernel [1]. Their work supports preemption but only on a single-core machine. They have not verified any assembly code nor connected their verified C-like source programs to any certified compiler so there is no end-to-end theorem about the entire kernel. They have not proved any progress properties so even their verified kernel modules or interrupt handlers could still diverge.

7 Conclusion

We have presented a novel extensible architecture for building certified concurrent OS kernels that have not only an efficient assembly implementation but also machine-checkable contextual correctness proofs. OS kernels developed using our layered methodology also come with a clean, rigorous, and layered specification of all kernel components. We show that building certified concurrent kernels is not only feasible but also quite practical. Our layered approach to certified concurrent kernels replaces the hardware-enforced "red line" with a large number of abstraction layers enforced via formal specification and proofs. We believe this will open up a whole new dimension of research efforts toward building truly reliable, secure, and extensible system software.

Acknowledgments

We would like to acknowledge the contribution of many former and current team members on various CertiKOS-related projects at Yale, especially Jérémie Koenig, Tahina Ramananandro, Shu-Chun Weng, Liang Gu, Mengqi Liu, Quentin Carbonneaux, Jan Hoffmann, Hernán Vanzetto, Bryan Ford, Haozhong Zhang, Yu Guo, and Joshua Lockerman. We also want to thank our shepherd Gernot Heiser and anonymous referees for helpful feedbacks that improved this paper significantly. This research is based on work supported in part by NSF grants 1065451, 1521523, and 1319671 and DARPA grants FA8750-12-2-0293, FA8750-16-2-0274, and FA8750-15-C-0082. Hao Chen's work is also supported in part by China Scholarship Council. The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright notation thereon. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of DARPA or the U.S. Government.

References

[1] The real-time kernel: μC/OS-II. http://micrium.com/rtos/ucosii, 1999 – 2012.

[2] The Coq proof assistant. http://coq.inria.fr, 1999 – 2016.

[3] The seL4 benchmark. https://github.com/smaccm/sel4bench-manifest, 2015.

[4] E. Alkassar, M. A. Hillebrand, W. J. Paul, and E. Petrova. Automated verification of a small hypervisor. In *Proc. 3rd International Conference on Verified Software: Theories, Tools, Experiments (VSTTE)*, pages 40–54, 2010.

[5] S. Amani, A. Hixon, Z. Chen, C. Rizkallah, P. Chubb, L. O'Connor, J. Beeren, Y. Nagashima, J. Lim, T. Sewell, J. Tuong, G. Keller, T. Murray, G. Klein, and G. Heiser. CoGENT: Verifying high-assurance file system implementations. In *Proc. 21st International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 175–188, 2016.

[6] T. Anderson. Private communication, Apr. 2016.

[7] T. Anderson and M. Dahlin. *Operating Systems Principles and Practice*. Recursive Books, 2011.

[8] A. Belay, A. Bittau, A. Mashtizadeh, D. Mazières, and C. Kozyrakis. Dune: Safe user-level access to privileged CPU features. In *Proc. 10th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 335–348, 2012.

[9] B. N. Bershad, S. Savage, P. Pardyak, E. G. Sirer, M. E. Fiuczynski, D. Becker, C. Chambers, and S. J. Eggers. Extensibility, safety and performance in the SPIN operating system. In *Proc. 15th ACM Symposium on Operating System Principles (SOSP)*, pages 267–284, 1995.

[10] W. R. Bevier. Kit: A study in operating system verification. *IEEE Transactions on Software Engineering*, 15(11):1382–1396, 1989.

[11] S. M. Blackburn, R. Garner, C. Hoffman, A. M. Khan, K. S. McKinley, R. Bentzur, A. Diwan, D. Feinberg, D. Frampton, S. Z. Guyer, M. Hirzel, A. Hosking, M. Jump, H. Lee, J. E. B. Moss, A. Phansalkar, D. Stefanović, T. VanDrunen, D. von Dincklage, and B. Wiedermann. The DaCapo benchmarks: Java benchmarking development and analysis. In *Proc. 21st ACM SIGPLAN conference on Object-Oriented Programing, Systems, Languages, and Applications (OOPSLA)*, pages 169–190. ACM Press, Oct. 2006.

[12] Q. Carbonneaux, J. Hoffmann, T. Ramananandro, and Z. Shao. End-to-end verification of stack-space bounds for C programs. In *Proc. 2014 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 270–281, 2014.

[13] H. Chen, D. Ziegler, T. Chajed, A. Chlipala, M. F. Kaashoek, and N. Zeldovich. Using Crash Hoare logic for certifying the FSCQ file system. In *Proc. 25th ACM Symposium on Operating System Principles (SOSP)*, pages 18–37, 2015.

[14] H. Chen, X. Wu, Z. Shao, J. Lockerman, and R. Gu. Toward compositional verification of interruptible OS kernels and device drivers. In *Proc. 2016 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 431–447, 2016.

[15] E. Cohen, M. Dahlweid, M. Hillebrand, D. Leinenbach, M. Moskal, T. Santen, W. Schulte, and S. Tobies. VCC: A practical system for verifying concurrent C. In *Proc. 22nd International Conference on Theorem Proving in Higher Order Logics (TPHOLs)*, pages 23–42, 2009.

[16] D. Costanzo, Z. Shao, and R. Gu. End-to-end verification of information-flow security for C and assembly programs. In *Proc. 2016 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 648–664, 2016.

[17] R. A. DeMillo, R. J. Lipton, and A. J. Perlis. Social processes and proofs of theorems and programs. In *Proc. 4th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 206–214, Jan. 1977.

[18] E. W. Dijkstra. The structure of the "THE"-multiprogramming system. *Communications of the ACM*, pages 341–346, May 1968.

[19] E. W. Dijkstra. Notes on structured programming. In O. J. Dahl, E. W. Dijkstra, and C. A. R. Hoare, editors, *Structured programming*, pages 1–82. Academic Press, 1972. ISBN 0-12-200550-3.

[20] D. R. Engler, M. F. Kaashoek, and J. O'Toole. Exokernel: An operating system architecture for application-level resource management. In *Proc. 15th ACM Symposium on Operating Systems Principles (SOSP)*, pages 251–266, Dec. 1995.

[21] X. Feng, Z. Shao, Y. Dong, and Y. Guo. Certifying low-level programs with hardware interrupts and preemptive threads. In *Proc. 2008 ACM SIGPLAN*

Conference on Programming Language Design and Implementation (PLDI), pages 170–182, 2008.

[22] I. Filipovic, P. W. O'Hearn, N. Rinetzky, and H. Yang. Abstraction for concurrent objects. *Theoretical Computer Science*, 411(51-52):4379–4398, 2010.

[23] R. Gu, J. Koenig, T. Ramananandro, Z. Shao, X. Wu, S.-C. Weng, H. Zhang, and Y. Guo. Deep specifications and certified abstraction layers. In *Proc. 42nd ACM Symposium on Principles of Programming Languages (POPL)*, pages 595–608, 2015.

[24] R. Gu, Z. Shao, X. Wu, J. Kim, J. Koenig, T. Ramananandro, V. Sjoberg, H. Chen, and D. Costanzo. Language and compiler support for building certified concurrent abstraction layers. Technical Report YALEU/DCS/TR-1530, Dept. of Computer Science, Yale University, New Haven, CT, October 2016.

[25] C. Hawblitzel, J. Howell, J. R. Lorch, A. Narayan, B. Parno, D. Zhang, and B. Zill. Ironclad apps: End-to-end security via automated full-system verification. In *Proc. 11th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 165–181, 2014.

[26] C. Hawblitzel, J. Howell, M. Kapritsos, J. R. Lorch, B. Parno, M. L. Roberts, S. T. V. Setty, and B. Zill. IronFleet: proving practical distributed systems correct. In *Proc. 25th ACM Symposium on Operating System Principles (SOSP)*, pages 1–17, 2015.

[27] G. Heiser. Private communication, September 2016.

[28] M. Herlihy and N. Shavit. *The Art of Multiprocessor Programming*. Morgan Kaufmann, Apr. 2008.

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

[30] G. C. Hunt and J. R. Larus. Singularity: rethinking the software stack. *Operating Systems Review*, 41 (2):37–49, 2007.

[31] M. F. Kaashoek. Parallel computing and the OS. In *Proc. SOSP History Day*, pages 10:1–10:35, 2015.

[32] C. Klein, J. Clements, C. Dimoulas, C. Eastlund, M. Felleisen, M. Flatt, J. McCarthy, J. Rafkind, S. Tobin-stadt, and R. B. Findler. Run your research: on the effectiveness of lightweight mechanization. In *Proc. 39th ACM Symposium on Principles of Programming Languages (POPL)*, pages 285–296, 2012.

[33] G. Klein, K. Elphinstone, G. Heiser, J. Andronick, D. Cock, P. Derrin, D. Elkaduwe, K. Engelhardt, R. Kolanski, M. Norrish, T. Sewell, H. Tuch, and S. Winwood. seL4: Formal verification of an OS kernel. In *Proc. 22nd ACM Symposium on Operating Systems Principles (SOSP)*, pages 207–220. ACM, 2009.

[34] G. Klein, J. Andronick, K. Elphinstone, T. Murray, T. Sewell, R. Kolanski, and G. Heiser. Comprehensive formal verification of an OS microkernel. *ACM Transactions on Computer Systems*, 32(1), Feb. 2014.

[35] B. W. Lampson. Experience with processes and monitors in Mesa. *Communications of the ACM*, 23 (2), Feb. 1980.

[36] D. Leinenbach and T. Santen. Verifying the Microsoft Hyper-V hypervisor with VCC. In *Proc. 2nd World Congress on Formal Methods*, pages 806–809, 2009.

[37] X. Leroy. The CompCert verified compiler. http://compcert.inria.fr/, 2005–2016.

[38] X. Leroy. Formal verification of a realistic compiler. *Communications of the ACM*, 52(7):107–115, 2009.

[39] W. Li, L. Gerard, and N. Shankar. Design and verification of multi-rate distributed systems. In *Proc. 2015 ACM/IEEE International Conference on Formal Methods and Models for Codesign (MEMOCODE)*, pages 20–29, 2015.

[40] H. Liang and X. Feng. A program logic for concurrent objects under fair scheduling. In *Proc. 43rd ACM Symposium on Principles of Programming Languages (POPL)*, pages 385–399, 2016.

[41] H. Liang, X. Feng, and M. Fu. A rely-guarantee-based simulation for verifying concurrent program transformations. In *Proc. 39th ACM Symposium on Principles of Programming Languages (POPL)*, pages 455–468, 2012.

[42] H. Liang, J. Hoffmann, X. Feng, and Z. Shao. Characterizing progress properties of concurrent objects via contextual refinements. In *Proc. 24th International Conference on Concurrency Theory (CONCUR)*, pages 227–241. Springer-Verlag, 2013.

[43] J. Liedtke. On micro-kernel construction. In *Proc. 15th ACM Symposium on Operating System Principles (SOSP)*, pages 237–250, 1995.

[44] N. A. Lynch and F. W. Vaandrager. Forward and backward simulations: I. Untimed systems. *Information and Computation*, 121(2):214–233, 1995.

[45] A. Madhavapeddy, R. Mortier, C. Rostos, D. Scott, B. Singh, T. Gazagnaire, S. Smith, S. Hand, and J. Crowcroft. Unikernels: library operating systems for the cloud. In *Proc. 18th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 461–472, 2013.

[46] J. M. Mellor-Crummey and M. L. Scott. Algorithms for scalable synchronization on shared-memory multiprocessors. *ACM Transactions on Computer Systems*, 9(1):21–65, Feb. 1991.

[47] D. Mosberger and T. Jin. Httperf - a tool for measuring web server performance. *SIGMETRICS Performance Evaluation Review*, 26(3):31–37, Dec. 1998. ISSN 0163-5999.

[48] P. G. Neumann, R. S. Boyer, R. J. Feiertag, K. N. Levitt, and L. Robinson. A provably secure operating system: its system, its applications, and proofs. Technical Report CSL-116, SRI Computer Science Laboratory, May 1980.

[49] W. Paul, M. Broy, and T. In der Rieden. The Verisoft XT Project. http://www.verisoftxt.de, 2010.

[50] T. Perrine, J. Codd, and B. Hardy. An overview of the kernalized secure operating system (KSOS). In *Proc. 7th DoD/NBS Computer Security Initiative Conference*, pages 146–160, Sep 1984.

[51] S. Peters, A. Danis, K. Elphinstone, and G. Heiser. For a microkernel, a big lock is fine. In *APSys '15 Asia Pacific Workshop on Systems, Tokyo, Japan*, 2015.

[52] R. Russell. VirtIO: Towards a de-facto standard for virtual I/O devices. *Operating System Review*, 42 (5):95–103, July 2008.

[53] Z. Shao. Certified software. *Communications of the ACM*, 53(12):56–66, December 2010.

[54] A. Vasudevan, S. Chaki, P. Maniatis, L. Jia, and A. Datta. überspark: Enforcing verifiable object abstractions for automated compositional security analysis of a hypervisor. In *Proc. 25th USENIX Security Symposium*, pages 87–104, 2016.

[55] M. von Tessin. *The Clustered Multikernel: An Approach to Formal Verification of Multiprocessor Operating-System Kernels*. PhD thesis, School of Computer Science and Engineering, The University of New South Wales, March 2013.

[56] J. R. Wilcox, D. Woos, P. Panchekha, Z. Tatlock, X. Wang, M. D. Ernst, and T. E. Anderson. Verdi: a framework for implementing and formally verifying distributed systems. In *Proc. 2015 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 357–368, 2015.

[57] F. Xu, M. Fu, X. Feng, X. Zhang, H. Zhang, and Z. Li. A practical verification framework for preemptive OS kernels. In *Proc. 28th International Conference on Computer-Aided Verification (CAV), Part II*, pages 59–79, 2016.

[58] J. Yang and C. Hawblitzel. Safe to the last instruction: automated verification of a type-safe operating system. In *Proc. 2010 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 99–110, 2010.

[59] N. Zeldovich, S. Boyd-Wickizer, E. Kohler, and D. Mazières. Making information flow explicit in HiStar. In *Proc. 7th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 263–278, 2006.

EbbRT: A Framework for Building Per-Application Library Operating Systems

Dan Schatzberg, James Cadden, Han Dong, Orran Krieger, Jonathan Appavoo
Boston University

Abstract

General purpose operating systems sacrifice per-application performance in order to preserve generality. On the other hand, substantial effort is required to customize or construct an operating system to meet the needs of an application. This paper describes the design and implementation of the Elastic Building Block Runtime (EbbRT), a *framework* for building per-application library operating systems. EbbRT reduces the effort required to construct and maintain library operating systems without hindering the degree of specialization required for high performance. We combine several techniques in order to achieve this, including a distributed OS architecture, a low-overhead component model, a lightweight event-driven runtime, and many language-level primitives. EbbRT is able to simultaneously enable performance specialization, support for a broad range of applications, and ease the burden of systems development.

An EbbRT prototype demonstrates the degree of customization made possible by our framework approach. In an evaluation of memcached, EbbRT and is able to attain 2.08× higher throughput than Linux. The node.js runtime, ported to EbbRT, demonstrates the broad applicability and ease of development enabled by our approach.

1 Introduction

Performance is a key concern for modern cloud applications. Even relatively small performance gains can result in significant cost savings at scale. The end of Dennard scaling and increasingly high-speed I/O devices has shifted the emphasis of performance to the CPU and, in turn, the software stack on which an application is deployed.

Existing general purpose operating systems sacrifice per-application performance in favor of generality. In response, there has been a renewed interest in library operating systems [28, 38], hardware virtualization [5, 44], and kernel bypass techniques [25, 48]. Common to these approaches is the desire to enable applications to interact with devices with minimal operating system involvement. This allows developers to customize the entire software stack to meet the needs of their application.

The problem with this approach is that it still requires significant engineering effort to implement the required system functionality. The consequence being that past systems have either targeted a narrow class of applications (e.g. packet processing) or, in order to be broadly applicable, constructed general purpose software stacks.

We believe that general purpose software stacks are subject to the same trade-off between performance and generality as existing commodity operating systems. In order to achieve high performance for a broad set of applications, we must bridge the gap between general purpose software, on which application development is easy, and the performance advantages obtained using customized, per-application software stacks where the development burden is high.

Our work addresses this gap with the Elastic Building Block Runtime (EbbRT), a *framework* for constructing per-application library operating systems. EbbRT reduces the effort required to construct and maintain library operating systems without restricting the degree of specialization required for high performance. We combine several techniques in order to achieve this.

1. EbbRT is comprised of a set of components, called Elastic Building Blocks (Ebbs), that developers can extend, replace, or discard in order to construct and deploy a particular application. This enables a greater degree of customization than a general purpose system and promotes the reuse of non-performance-critical components.

2. EbbRT uses a lightweight execution environment that allows application logic to directly interact with

hardware resources, such as memory and devices.

3. EbbRT applications are distributed across both specialized and general purpose operating systems. This allows functionality to be offloaded, which reduces the engineering effort required to port applications.

In this paper we describe the design and implementation of the EbbRT framework, along with several of its system components (e.g., a scalable network stack and a high-performance memory allocator). We demonstrate that library operating systems constructed using EbbRT outperform Linux on a series of compute and network intensive workloads. For example, a memcached port to EbbRT, run on a commodity hypervisor, is able to attain 2.08× greater throughput than memcached run on Linux. Additionally, we show that, with modest developer effort, EbbRT is able to support large, complex applications. In particular, node.js, a managed runtime environment, was ported in two weeks by a single developer.

The remainder of the paper is structured as follows: section 2 outlines the objectives of EbbRT, section 3 presents the high-level architecture and design of our framework, section 4 describes the implementation, section 5 presents the evaluation of EbbRT, section 6 discusses related work, and section 7 concludes.

2 Objectives

The following three objectives guide our design and implementation:

Performance Specialization: To achieve high performance we seek to allow applications to efficiently specialize the system at every level. To facilitate this, we provide an event-driven execution environment with minimal abstraction over the hardware; EbbRT applications can provide event handlers to directly serve hardware interrupts. Also, our Ebb component model has low enough overhead to be used throughout performance-sensitive paths, while also enabling compiler optimizations, such as aggressive inlining.

Broad Applicability: To ensure high utility, a framework should support a broad set of applications. EbbRT is designed and implemented to support the rich set of existing libraries and complex managed runtimes on which applications depend. We adopt a heterogeneous distributed architecture, called the MultiLibOS [49] model, wherein EbbRT library operating systems run alongside general purpose operating systems and offload functionality transparently. EbbRT library operating systems can be integrated with a process of the general purpose OS.

Ease of Development: We strive to make the development of application-specific systems easy. EbbRT exploits modern language techniques to simplify the task of writing new system software, while the Ebb model provides an abstraction to encapsulate existing system components. The barrier to porting existing applications is lowered through the use of function offloading between an EbbRT library OS and a general purpose OS.

Attaining all three of these objectives simultaneously is a challenging but critical step towards bridging the gap between general purpose and highly specialized software stacks.

3 System Design

This section describes the high-level design of EbbRT. In particular the three elements of the design discussed are: 1. a heterogeneous distributed structure, 2. a modular system structure, and 3. a non-preemptive event-driven execution environment.

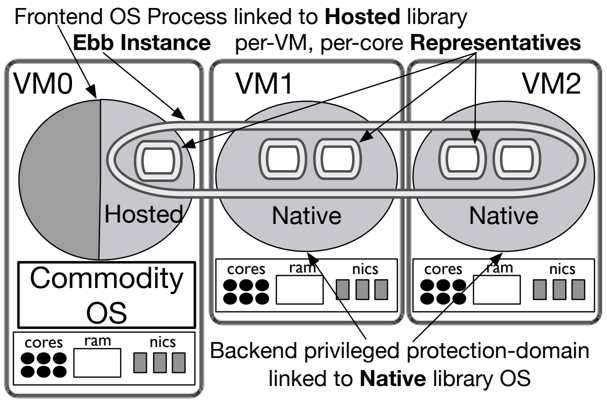

Frontend OS Process linked to **Hosted** library
Ebb Instance
per-VM, per-core **Representatives**
VM0
VM1
VM2
Hosted
Native
Native
Commodity OS
cores
ram
nics
Backend privileged protection-domain linked to **Native** library OS

Figure 1: High Level EbbRT architecture

3.1 Heterogeneous Distributed Structure

Our design is motivated in-part by the common deployment strategies of cloud applications. Infrastructure as a Service enables a single application to be deployed across multiple machines within an isolated network. In this context, it is not necessary to run general purpose operating systems on all the machines of a deployment. Rather, an application can be deployed across a heterogeneous mix of specialized library OSs and general purpose operating systems as illustrated in Figure 1.

To facilitate this deployment model, EbbRT is implemented as both a lightweight bootable runtime and a user-level library that can be linked into a process of a general purpose OS [49]. We refer to the bootable library

OS as the *native* runtime and the user-level library as the *hosted* runtime.

The native runtime allows application software to be written directly to hardware interfaces uninhibited by legacy interfaces and protection mechanisms of a general purpose operating system. The native runtime sets up a single address space, basic system functionality (e.g. timers, networking, memory allocation) and invokes an application entry point, all while running at the highest privilege level. The EbbRT design depends on application isolation at the network layer, either through switch programming or virtualization, making it amenable to both virtualized and bare-metal environments.

The hosted user-space library allows EbbRT applications to integrate with legacy software. This frees the native library OSs from the burden of providing compatibility with legacy interfaces. Rather, functionality can be offloaded via communication with the hosted environment.

A common deployment of a EbbRT application consists of a hosted process and one or more native runtime instances communicating across a local network. A user is able to interact with the EbbRT application through the hosted runtime, as they would any other process of the general purpose OS, while the native runtime supports the performance-critical portion of the application.

3.2 Modular System Structure

To provide a high degree of customization, EbbRT enables application developers to modify or extend all levels of the software stack. To support this, EbbRT applications are almost entirely comprised of objects we call *Elastic Building Blocks* (Ebbs). As with objects in many programming languages, Ebbs encapsulate implementation details behind a well-defined interface.

Ebbs are distributed, multi-core fragmented objects [9, 39, 51], where the namespace of Ebbs is shared across both the native and hosted runtimes. Figure 1 illustrates an Ebb spanning both hosted and native runtimes of an application. An EbbRT application typically consists of multiple Ebb instances. The framework is composed of base Ebb types that a developer can use to construct an EbbRT application.

When an Ebb is invoked, a local *representative* handles the call. Representatives may communicate with each other to satisfy the invocation. For example, an object providing file access might have representatives on a native instance simply function-ship requests to a hosted representative which translates these requests into requests on the local file system. By encapsulating the distributed nature of the object, optimizations such as RDMA, caching, using local storage, etc. would all be hidden from clients of the filesystem Ebb.

Ebb reuse is critical to easing development effort. Exploiting modularity promotes reuse and evolution of the EbbRT framework. Developers can build upon the Ebb structure to provide additional libraries of components that target specific application use cases.

3.3 Execution Model

Execution in EbbRT is non-preemptive and event-driven. In the native runtime there is one event loop per core which dispatches both external (e.g. timer completions, device interrupts) and software generated events to registered handlers. This model is in contrast to a more standard threaded environment where preemptable threads are multiplexed across one or more cores. Our non-preemptive event-driven execution model provides a low overhead abstraction over the hardware. This allows our implementation to directly map application software to device interrupts, avoiding the typical costs of scheduling decisions or protection domain switches.

EbbRT provides an analogous environment within the hosted library by providing an event loop using underlying OS functionality such as `poll` or `select`. While the hosted environment cannot achieve the same efficiency as our native runtime, we provide a compatible environment to allow software libraries to be reused across both runtimes.

Many cloud applications are driven by external requests such as network traffic so the event-driven programming environment provides a natural way to structure the application. Indeed, many cloud applications use a user-level library (e.g. libevent [46], libuv [34], Boost ASIO [29]) to provide such an environment.

However, asynchronous, event-driven programming may not be a good fit for all applications. To this end, we provide a simple cooperative threading model on top of events. This allows for blocking semantics and a concurrency model similar to the Go programming language. We discuss support for long running events further in Section 4.2.

The non-preemptive event execution, along with support for cooperative threading, allows the native runtime to be lightweight yet provides sufficient flexibility for a wide range of applications. Such qualities are critical in enabling performance specialization without sacrificing applicability.

4 Implementation

In this section we provide an overview of the system software and then describe details of the implementation.

		Primitives			**External Libraries**				
		Futures	Lambdas	IOBufs	std c++	Boost	Intel TBB	capnproto	**Description**
Memory	PageAllocator				✓	✓	✓		Power of two physical page frame allocator
	VMemAllocator				✓				Allocates virtual address space
	SlabAllocator				✓	✓			Allocates fixed sized objects
	GeneralPurposeAllocator				✓				General purpose memory allocator
Objects	EbbAllocator				✓	✓			Allocates EbbIds
	LocalIdMap				✓	✓	✓		Local data store for Ebb data and fault resolution
	GlobalIdMap	✓		✓	✓			✓	Application-wide data store for Ebb data
Event	EventManager	✓	✓		✓	✓			Creates events and manages hardware interrupts
	Timer				✓	✓			Delay based scheduling of events
I/O	NetworkManager	✓	✓	✓	✓	✓			Implements TCP/IP stack
	SharedPoolAllocator				✓	✓			Allocates network ports
	NodeAllocator	✓			✓	✓		✓	Allocates, configures, and releases IAAS resources
	Messenger	✓	✓		✓				Cross node Ebb to Ebb communication
	VirtioNet			✓	✓				VirtIO network device driver

Table 1: The core Ebbs that make up EbbRT. A gray row indicates that the Ebb has a multi-core implementation (one representative per core) while the others use a single shared representative.

4.1 Software Structure Overview

EbbRT is comprised of an x86_64 library OS and toolchain as well as a Linux userspace library. Both runtimes are written predominately in C++14 totaling 14,577 lines of new code [59]. The native library is packaged with a GNU toolchain (gcc, binutils, libstdc++) and libc (newlib) modified to support a x86_64-ebbrt build target. Application code compiled with the toolchain will produce a bootable ELF binary linked with the library OS. We provide C and C++ standard library implementations which make it straightforward to use many third party software libraries as shown in Table 1. The support and use of standard software libraries to implement system-level functionality makes it much easier for library and application developers to understand and modify system-level Ebbs.

We chose not to strive for complete Linux or POSIX compatibility. We feel that enforcing compatibility with existing OS interfaces would be restrictive and, given the function offloading enabled by our heterogeneous distributed structure, unnecessary. Rather, we provide minimalist interfaces above the hardware, which allows for a broad set of software to be developed on top.

EbbRT provides the necessary functionality for events to execute and Ebbs to be constructed and used. This entails functionality such as memory management, networking, timers, and I/O. This functionality is provided by the core system Ebbs shown in Table 1

4.2 Events

Both the hosted and native environments provide an event driven execution model. Within the hosted environment we use the Boost ASIO library [29] in order to interface with the system APIs. Within the native environment, our event-driven API is implemented directly on top of the hardware interfaces. Here, we focus our description on the implementation of events within the native environment.

When the native environment boots, an event loop per core is initialized. Drivers can allocate a hardware interrupt from the `EventManager` and then bind a handler to that interrupt. When an event completes and the next hardware interrupt fires, a corresponding exception handler is invoked. Each exception handler execution begins on the top frame of a per-core stack. The exception handler checks for an event handler bound to the corresponding interrupt and then invokes it. When the event handler returns, interrupts are enabled and more events can be processed. Therefore events are non-preemptive and typically generated by a hardware interrupt.

Applications can invoke synthetic events on any core in the system. The `Spawn` method of the

EventManager receives an event handler, which is later invoked from the event loop. Events invoked with Spawn are only executed once. A handler for a reoccurring event can be installed as an IdleHandler.

In order to prevent interrupt starvation, when an event completes the EventManager, 1. enables then disables interrupts, handling any pending interrupts, 2. dispatches a single synthetic event, 3. invokes all IdleHandlers and then 4. enables interrupts and halts. If any of these steps result in an event handler being invoked, then the process starts again at the beginning. This way, hardware interrupts and synthetic events are given priority over repeatedly invoked IdleHandlers.

While our EventManager implementation is simple, it provides sufficient functionality to achieve interesting dynamic behavior. For example, our network card driver implements adaptive polling in the following way: an interrupt is allocated from the EventManager and the device is programmed to fire that interrupt when packets are received. The event handler will process each received packet to completion before returning control to the event loop. If the interrupt rate exceeds a configurable threshold, the driver disables the receive interrupt and installs an IdleHandler to process received packets. The EventManager will repeatedly call the idle handler from within the event loop, effectively polling the device for more data. When the packet arrival rate drops below a configurable threshold, the driver re-enables the receive interrupt and disables the idle handler, returning to interrupt-driven packet processing.

Given our desire to enable reuse of existing software, we adopt a cooperative threading model which allows events to explicitly save and restore control state (e.g., stack and volatile register state). At the point where the block would occur, the current event saves its state and relinquishes control back to the event loop, where the processing of pending events is resumed. The original event state can be restored, and its execution resumed, when the asynchronous work completes. The save and restore event mechanisms enable explicit cooperative scheduling between events, facilitating familiar blocking semantics. This has allowed us to quickly port software libraries that require blocking system calls.

A limitation of non-preemptive execution is the difficulty of mapping long-running threads with no I/O to an event-driven model. If the processor is not yielded periodically, event starvation can occur. At present we do not provide a completely satisfactory solution. Building a preemptive scheduler on top of events would be possible, though we fear it would fragment the set of Ebbs into those that depend on non-preemptive execution and those that don't. Alternatively, we have discussed dedicating processors to executing these long-running threads and therefore avoiding any starvation issues, similar to IX [5]. Nonetheless, we have not run into this problem in practice; most cloud applications rely heavily on I/O, and concern for starvation is reduced as we only support the execution of a single process.

4.3 Elastic Building Blocks

Nearly all software in EbbRT is written as elastic building blocks, which encapsulate both the data and function of a software component. Ebbs hide from clients the distributed or parallel nature of objects and can be extended or replaced for customization. An Ebb provides an interface using a standard C++ class definition. Every instance of an Ebb has a system-wide unique EbbId (32 bits in our current implementation). Software invokes the Ebb by converting the EbbId into an EbbRef which can be dereferenced to a per-core *representative* which is a reference to an instance of the underlying C++ class. We use C++ templates to implement the EbbRef generically for all Ebb classes.

Ebbs may be invoked on any machine or core within the application. Therefore, it is necessary for initialization of the per-core representatives to happen on-demand to mitigate initialization overheads for short-lived Ebbs. An EbbId provides an offset into a virtual memory region backed with distinct per-core pages which holds a pointer to the per-core representative (or NULL if it does not exist). When a function is called on an EbbRef, it checks the per-core representative pointer — in the common case where it is non-null, it is dereferenced and the call is made on the per-core representative. If the pointer is null, then a type specific *fault handler* is invoked which must return a reference to a representative to be called or throw a language-level exception. Typically, a fault handler will construct a representative and store it in the per-core virtual memory region so future invocations will take the fast-path. Our hosted implementation of Ebb dereferences uses per-thread hash-tables to store representative pointers.

The construction of a representative may require communication with other representatives either within the machine or on other machines. EbbRT provides core Ebbs that support distributed data storage and messaging services. These facilities span and enable communication between the EbbRT native and hosted instances and utilize network communication as needed.

Ebb modularity is both flexible and efficient, making them suitable for high-performance components. Previous systems providing a partitioned object model either used relatively heavy weight invocation across a distributed system [56], or more efficient techniques constrained to a shared memory system [17, 30]. Ebbs are unique in their ability to accommodate both use cases.

The fast-path cost of an Ebb invocation is one predictable conditional branch and one unconditional branch more than a normal C++ object dereference. Additionally, our use of static dispatch (EbbRef's are templated by the representative's type) enables compiler optimizations such as function inlining.

We intentionally avoided using interface definition languages such as COM [60], CORBA [57], or Protocol Buffers [21]. Our concern was that these often require serialization and deserialization at all interface boundaries, which would promote much coarser grained objects than we desire. Our ability to use native C++ interfaces allows Ebbs to pass complex data structures amongst each other. This also necessitates that all Ebb invocations be local. Ebb representative communication is encapsulated and may internally serialize data structures as needed.

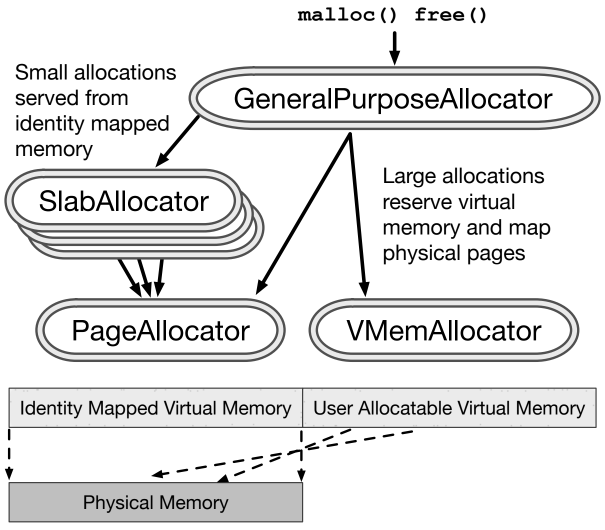

malloc() free()
Small allocations served from identity mapped memory
GeneralPurposeAllocator
SlabAllocator
Large allocations reserve virtual memory and map physical pages
PageAllocator
VMemAllocator
Identity Mapped Virtual Memory
User Allocatable Virtual Memory
Physical Memory

Figure 2: Memory Management Ebbs

4.4 Memory Management

Memory allocation is a performance critical facet of many cloud applications, and our focus on short-lived events puts increased pressure on the memory allocator to perform well. Here we present our default native memory allocator and highlight aspects of it which demonstrate the synergy of the elements in the EbbRT design. Figure 2 illustrates EbbRT's memory management infrastructure and the virtual and physical address space relationships.

The EbbRT memory allocation subsystem is similar to that of the Linux Kernel. The lowest-level allocator is the PageAllocator, which allocates power of two sized pages of memory. Our default PageAllocator implementation uses buddy-allocators, one per NUMA node. On top of the PageAllocator are the SlabAllocator Ebbs, which are used to allocate fixed size objects. Our default SlabAllocator implementation uses per-core and per-NUMA node representatives to store object free-lists and partially allocated pages. This design is based on the Linux Kernel's SLQB allocator [12]. The GeneralPurposeAllocator, invoked via malloc, is implemented using multiple SlabAllocator instances, each responsible for allocating objects of a different fixed size. To serve a request, the GeneralPurposeAllocator invokes the SlabAllocator with the closest size greater or equal to the requested size. For requests exceeding the largest SlabAllocator size, the GeneralPurposeAllocator will allocate a virtual memory region mapped in from the VMemAllocator and backed by pages mapped in from the PageAllocator.

By defining the memory allocators as Ebbs, we allow any one of the components to be replaced or modified without impacting the others. In addition, because our implementation uses C++ templates for static dispatch, the compiler is able to optimize calls across Ebb interfaces. For example, calls to malloc that pass a size known at compile time are optimized to directly invoke the correct SlabAllocator within the GeneralPurposeAllocator.

A key property of memory allocations in EbbRT is that most allocations are serviced from identity mapped physical memory. This applies to all allocations made by the GeneralPurposeAllocator that do not exceed the largest SlabAllocator size (virtual mappings are used for larger allocations). Identity mapped memory allows application software to perform zero-copy I/O with standard allocations rather than needing to allocate memory specifically for DMA.

Another benefit of the EbbRT design is that, due to the lack of preemption, most allocations can be serviced from a per-core cache without any synchronization. Avoiding atomic operations is so important that high performance allocators like TCMalloc [19] and jemalloc [14] use per-*thread* caches to do so. These allocators then require complicated algorithms to balance the caching across a potentially dynamic set of threads. In contrast, the number of cores is typically static and generally not too large, simplifying EbbRT's balancing algorithm.

While a portion of the virtual address space is reserved to identity map physical memory and some virtual memory is used to provide per-core regions for Ebb invocation, the vast majority of the virtual address space is available for application use. Applications can allocate virtual regions by invoking the VMemAllocator and passing in a handler to be invoked on faults to that allocated region. This allows applications to implement

```
 1  // Sends out an IPv4 packet over Ethernet
 2  Future<void> EthIpv4Send(uint16_t eth_proto, const Ipv4Header& ip_hdr, IOBuf buf) {
 3    Ipv4Address local_dest = Route(ip_hdr.dst);
 4    Future<EthAddr> future_macaddr = ArpFind(local_dest);   /* asynchronous call   */
 5    return future_macaddr.Then(
 6      // continuation is passed in as an argument
 7      [buf = move(buf), eth_proto](Future<EthAddr> f) {     /* lambda definition   */
 8        auto& eth_hdr = buf->Get<EthernetHeader>();
 9        eth_hdr.dst = f.Get();
10        eth_hdr.src = Address();
11        eth_hdr.type = htons(eth_proto);
12        Send(move(buf));
13      });                                                   /* end of Then() call  */
14  }
```

Figure 3: Network code path to route and send and Ethernet frame.

arbitrary paging policies.

Our memory management demonstrates some of the advantages provided by EbbRT's design. First, we use Ebbs to create per-core representatives for multi-core scalability and to provide encapsulation to enable the different allocators to be replaced. Second, the lack of preemption enables us to use the per-core representatives without synchronization. Third, the library OS design enables tighter collaboration between system components and application components — as exemplified by the application's ability to directly manage virtual memory regions and achieve zero-copy interactions with device code.

4.5 Lambdas and Futures

One of the core objectives of our design is mitigating complexity to ease development. Critics of event-driven programming point out several properties which place increased burden on the developer.

One concern is that event-driven programming tends to obfuscate the control flow of the application [58]. For example, a call path that requires the completion of an asynchronous event will often pass along a callback function to be invoked when the event completes. The callback is invoked within a context different than that of the original call path, so it falls on the programmer to construct *continuations*, i.e. control mechanisms used to save and restore state across invocations. C++ has recently added support for anonymous inline functions called *lambdas*. Lambdas can capture local state that can be referred to when the lambda is invoked. This removes the burden of manually saving and restoring state, and makes code easier to follow. We use lambdas in EbbRT to alleviate the burden of constructing continuations.

Another concern with event-driven programming is that error handling is much more complicated. The predominant mechanism for error handling in C++ is exceptions. When an error is encountered, an exception is thrown and the stack unwound to the most recent try/-catch block, which will handle the error. Because event-driven programming splits one logical flow of control across multiple stacks, exceptions must be handled at every event boundary. This puts the burden on the developer to catch exceptions at additional points in the code and either handle them or forward them to an error handling callback.

Our solution to these problems is our implementation of *monadic futures*. Futures are a data type for asynchronously produced values, originally developed for use in the construction of distributed systems [37]. Figure 3 illustrates a code path in the EbbRT network stack that utilizes lambdas and futures to route and send an Ethernet frame. The `ArpFind` function (line 4) translates an IP address to the corresponding MAC address either through a lookup into the ARP cache or by sending out an ARP request to be processed asynchronously. In either case, `ArpFind` returns a `Future<EthAddr>`, which represents the future result of the ARP translation. A future cannot be directly operated on. Instead, a lambda can be applied to it using the `Then` method (line 5). This lambda is invoked once the future is fulfilled. When invoked, the lambda receives the fulfilled future as a parameter and can use the `Get` method to retrieve its underlying value (line 9). In the event that the future is fulfilled before the `Then` method is invoked (for example, `ArpFind` retrieves the translation directly from the ARP cache) the lambda is invoked synchronously.

The `Then` method of a future returns a new future representing the value to be returned by the applied function, hence the term monadic. This allows other software components to chain further functions to be invoked on completion. In this example, the `EthIpv4Send` method returns a `Future<void>` which merely represents the completion of some action and provides no data.

Futures also aid in error processing. Each time `Get` is invoked, the future may throw an exception representing a failure to produce the value. If not explicitly handled, the future returned by `Then` will hold this exception instead of a value. The only invocation of `Then` that must handle the error is the final one, any intermediate exceptions will naturally flow to the first function which attempts to catch the exception. This behavior mirrors the behavior of exceptions in synchronous code. In this example, any error in ARP resolution will be propagated to the future returned by `EthIpv4Send` and handled by higher-level code.

C++ has an implementation of futures in the standard library. Unlike our implementation, it provides no `Then` function, necessary for chaining callbacks. Instead users are expected to block on a future (using `Get`). Other languages such as C# and JavaScript provide monadic futures similar to ours.

As seen in Table 1, futures are used pervasively in interface definitions for Ebbs, and lambdas are used in place of more manual continuation construction. Our experience using lambdas and futures has been positive. Initially, some members of our group had reservations about using these unfamiliar primitives as they hide a fair amount of potentially performance sensitive behavior. As we have gained more experience with these primitives, it has been clear that the behavior they encapsulate is common to many cases. Futures in particular encapsulate sometimes subtle synchronization code around installing a callback and providing a value (potentially concurrently). While this code has not been without bugs, we have more confidence in its correctness based on its use across EbbRT.

4.6 Network Stack

We originally looked into porting an existing network stack to EbbRT. However, we eventually implemented a new network stack for the native environment, providing IPv4, UDP/TCP, and DHCP functionality in order to provide an event-driven interface to applications, minimize multi-core synchronization, and enable pervasive zero-copy. The network stack does not provide a standard BSD socket interface, but rather enables tighter integration with the application to manage the resources of a network connection.

During the development of EbbRT we found it necessary to create a common primitive for managing data that could be received from or sent to hardware devices. To support the development of zero-copy software, we created the `IOBuf` primitive. An `IOBuf` is a descriptor which manages ownership of a region of memory as well as a *view* of a portion of that memory. Rather than having applications explicitly invoke `read` with a buffer to be populated, they install a handler which is passed an `IOBuf` containing network data for their connection. This `IOBuf` is passed synchronously from the device driver through the network stack. The network stack does not provide any buffering, it will invoke the application as long as data arrives. Likewise, the interface to send data accepts a chain of `IOBufs` which can use scatter/gather interfaces.

Most systems have fixed size buffers in the kernel which are used to pace connections (e.g. manage TCP window size, cause UDP drops). In contrast, EbbRT allows the application to directly manage its own buffering. In the case of UDP, an overwhelmed application may have to drop datagrams. For a TCP connection, an application can explicitly set the window size to prevent further sends from the remote host. Applications must also check that outgoing TCP data fits within the currently advertised sender window before telling the network stack to send it or buffer it otherwise. This allows the application to decide whether or not to delay sending to aggregate multiple sends into a single TCP segment. Other systems typically accomplish this using Nagle's algorithm which is often associated with poor latency [41]. An advantage of EbbRT's approach to networking is the degree to which an application can tune the behavior of its connections at runtime. We provide default behaviors which can be inherited from for those applications which do not require this degree of customization.

One challenge with high-performance networking is the need to synchronize when accessing connection state [47]. EbbRT stores connection state in an RCU [40] hash table which allows common connection lookup operations to proceed without any atomic operations. Due to the event-driven execution model of EbbRT, RCU is a natural primitive to provide. Because we lack preemption, entering and exiting RCU critical sections are free. Connection state is only manipulated on a single core which is chosen by the application when the connection is established. Therefore, common case network operations require no synchronization.

The EbbRT network stack is an example of the degree of performance specialization our design enables. By involving the application in network resource management, the networking stack avoids significant complexity. Historically, network stack buffering and queuing has been a significant factor in network performance. EbbRT's design does not solve these problems, but instead enables applications to more directly control these properties and customize the system to their characteristics. The zero-copy optimization illustrates the value of having all physical memory identity mapped, unpaged, and within a single address space.

5 Evaluation

Through evaluating EbbRT we aim to affirm that our implementation fulfills the following three objectives discussed in Section 2: 1. supports high-performance specialization, 2. provides support for a broad set of applications, and 3. simplifies the development of application-specific systems software.

We run our evaluations on a cluster of servers connected via a 10GbE network and commodity switch. Each machine contains two 6-core Xeon E5-2630L processors (run at 2.4 GHz), 120 GB of RAM, and an Intel X520 network card (82599 chipset). The machines have been configured to disable Turbo Boost, hyper-threads, and dynamic frequency scaling. Additionally, we disable IRQ balancing and explicitly assign NIC IRQ affinity. For the evaluation, we pin each application thread to a dedicated physical core.

Each machine boots Ubuntu 14.04 (trusty) with Linux kernel version 3.13. The EbbRT native library OSs are run as virtual machines, which are deployed using QEMU (2.5.0) and the KVM kernel module. In addition, the VMs use a `virtio-net` paravirtualized network card with support of the `vhost` kernel module. We enable multiqueue receive flow steering for multicore experiments. Unless otherwise stated, all Linux applications are run within a similarly configured VM and on the same OS and kernel version as the host.

The evaluations are broken down as follows: 1. micro-benchmarks designed to quantify the base overheads of the primitives in our native environment and 2. macro-benchmarks that exercise EbbRT in the context of real applications. While the EbbRT hosted library is a primary component of our design, it is not intended for high-performance, but rather to facilitate the integration of functionality between a general purpose OS process and native instances of EbbRT. Therefore, we focus our evaluation on the EbbRT native environment.

5.1 Microbenchmarks

The first micro-benchmark evaluates the memory allocator and aims to establish that the overheads of our Ebb mechanism do not preclude the construction of high-performance components. The second set of micro-benchmarks evaluate the latencies and throughput of our network stack and exercise several of the system features we've discussed, including idle event processing, lambdas, and the `IOBuf` mechanism.

5.1.1 Memory Allocation

In K42 [30], we did not define its memory allocator as a fragmented object because the invocation overheads (e.g., virtual function dispatch) were thought to be too expensive. A goal for the design of our Ebb mechanism is to provide near-zero overhead so that all components of the system can be defined as Ebbs.

The costs of managing memory is critical to the overall performance of an application. Indeed, custom memory allocators have shown substantial improvements in application performance [7]. We've ported *threadtest* from the Hoard [6] benchmark suite to EbbRT in order to compare the performance of the default EbbRT memory allocator to that of the glibc 2.2.5 and jemalloc 4.2.1 allocators.

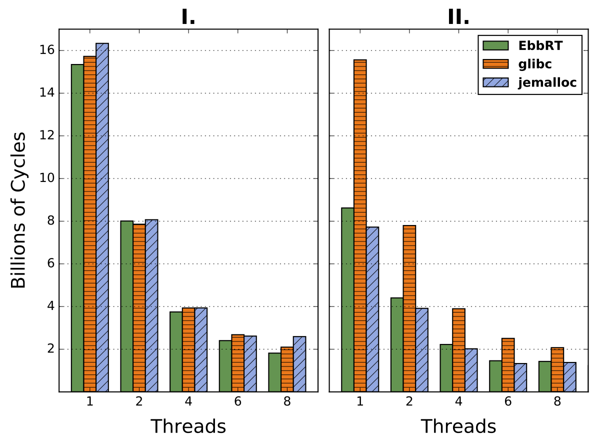


Figure 4: Hoard Threadtest. Y-axis represents threads, t. I.) N=100,000, i=1000; II.) N=100, i=1,000,000.

In *threadtest*, each thread t allocates and frees $\frac{N}{t}$ 8 byte objects. This task is repeated for i iterations. Figure 4 shows the cycles required to complete the workload across varying amounts of threads. We run *threadtest* in two configurations. In configuration *I.*, the number of objects, N, is large, while the number of iterations is small. In configuration *II.* the number of objects is smaller and the iteration count is increased. The total number of memory operations is the same across both configurations.

In the figure we see EbbRT's memory allocator scales competitively with the production allocators. Our scalability advantage is in part due to locality enabled by the per-core Ebb representatives of the memory allocator and our lack of preemption which remove any synchronization requirements between representatives. The jemalloc allocator achieves similar scalability benefits by avoiding synchronization through the use of per-thread caches.

This comparison is not intended to establish the EbbRT memory allocator to be the best in all situations, nor is it an exhaustive memory allocator study. Rather, we aim to demonstrate that the overheads of the Ebb mechanism do not preclude us from the construction of high-performance components.

5.1.2 Network Stack

To evaluate the performance of our network stack we ported the NetPIPE [52] and iPerf [54] benchmarks to EbbRT. NetPIPE is a popular ping-pong benchmark where a client sends a fixed-size message to the server, which is then echoed back after being completely received. In the iPerf benchmark, a client opens a TCP stream and sends fixed-size messages which the server receives and discards. With small message sizes, the NetPIPE benchmark illustrates the latency of sending and receiving data over TCP. The iPerf benchmark confirms that our run-to-completion network stack doesn't preclude high throughput applications. An EbbRT iPerf server was shown to saturate our 10GbE network with a stream of 1 kB message sizes.

Figure 5 shows NetPIPE goodput achieved as a function of message size. Two EbbRT servers achieve a one-way latency of 24.53 µs for 64 B message sizes and are able to attain 4 Gbps of goodput with messages as small as 100 kB. In contrast, two Linux VMs achieve a one-way latency of 34.27 µs for 64 B message sizes and required 200 kB sized messages to achieve equivalent goodput.

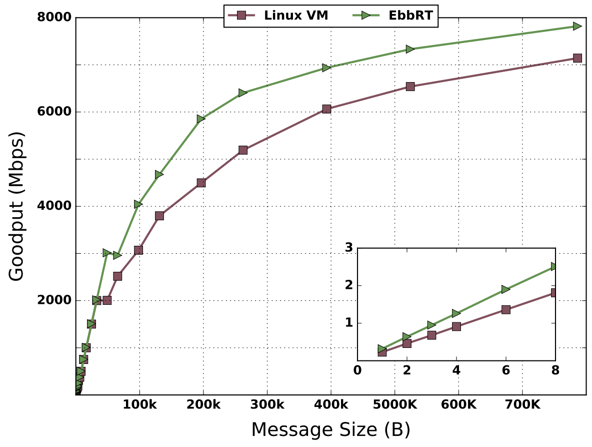


Figure 5: NetPIPE performance as a function of message size. Inset shows small message sizes.

With small messages, both systems suffer some additional latency due to hypervisor processing involved in implementing the paravirtualized NIC. However, EbbRT's short path from (virtual) hardware to application achieves a 40% improvement in latency with NetPIPE. This result illustrates the benefits of a non-preemptive event-driven execution model and zero-copy instruction path. With large messages, both systems must suffer a copy on packet reception due to the hypervisor, but EbbRT does no further copies, whereas Linux must copy to user-space and then again on transmission.

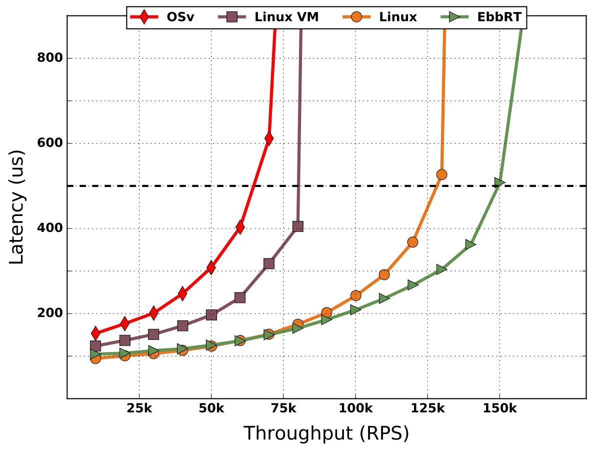


Figure 6: Memcached Single Core Performance

This explains the difference in Netpipe goodput before the network becomes the bottleneck.

5.2 Memcached

We evaluate memcached [15], an in-memory key-value store that has become a common benchmark in the examination and optimization of networked systems. Previous work has shown that memcached incurs significant OS overhead [27], and hence is a natural target for OS customization. Rather than port the existing memcached and associated event-driven libraries to EbbRT we re-implemented memcached, writing it directly to the EbbRT interfaces.

Our memcached implementation is a multi-core application that supports the standard memcached binary protocol. In our implementation, TCP data is received synchronously from the network card and passed up to the application. The application parses the client request and constructs a reply, which is sent out synchronously. The entire execution path, up to the application and back again, is run without pre-emption. Key-value pairs are stored in an RCU hash table to alleviate lock contention, a common cause for poor scalability in memcached. Our implementation of memcached totals 361 lines of code. We lack some features of the standard memcached (namely authentication and some of per-key commands such as queue operations), but are otherwise protocol compatible. Functionality support has been added incrementally as needed by our workloads.

We compare our EbbRT implementation of memcached, run within a VM, to the standard implementation (v.1.4.22) run within a Linux VM, and as a Linux process run natively on our machine. We use the `mutilate` [31] benchmarking tool to place a particular load on the server and measure response latency. We configure

	Request/sec	Inst/cycle	Inst/request	LLC ref/cycle	I-cache miss/cycle
EbbRT	379387	0.81	5557	0.0081	0.0079
Linux VM	137194	0.71	13604	0.0098	0.0339

Table 2: Memcached CPU-efficiency metrics

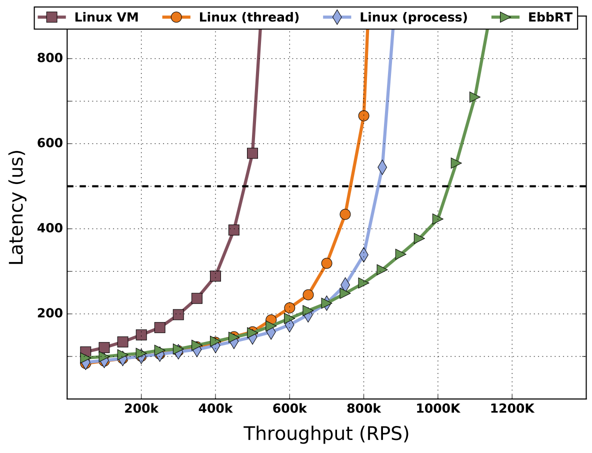


Figure 7: Memcached Multicore Performance

`mutilate` to generate load representative of the Facebook ETC workload [2], which has 20 B–70 B keys and most values sized between 1 B–1024 B. All requests are issued as separate memcached requests (no `multiget`) over TCP. The client is configured to pipeline up to four requests per TCP connection. We dedicate 7 machines to act as load-generating clients for a total of 664 connections per server.

Figure 6 presents the 99th percentile latency as a function of throughput for single core memcached servers. At a 500 µs 99th percentile Service Level Agreement (SLA), single core EbbRT is able to attain 1.88× higher throughput than Linux within a VM. EbbRT outperforms Linux running natively by 1.15×, even with the hypervisor overheads incurred. Additionally, we evaluated the performance of OS^v [28], a general purpose library OS that similarly targets cloud applications run in a virtualized environment. OS^v differs from EbbRT by providing a Linux ABI compatible environment, rather than supporting a high-degree of specialization. We found that the performance of `memcached` on OS^v was not competitive with either Linux or EbbRT with a single core. Additionally, OS^v's performance degrades when scaled up to six cores (omitted from figure 7) due to a lack of multiqueue support in their `virtio-net` device driver.

Figure 7 presents the evaluation of memcached running across six cores. At a 500 µs 99th percentile SLA, six core EbbRT is able to attain a 2.08× higher throughput than Linux within a VM and 1.50× higher than Linux native. To eliminate the performance impact of application-level contention, we also evaluated memcached run natively as six separate processes, rather than a single multithreaded process ("Linux (process)" in Figure 7). EbbRT outperforms the multiprocess memcached by 1.30× at 500 µs 99th percentile SLA.

To gain insight into the source of EbbRT's performance advantages, we examine the CPU-efficiency of the memcached servers. We use the Linux Kernel `perf` utility to gather data across a 10 second duration of a fully-loaded single core memcached server run within a VM. Table 2 presents these statistics. We see that the EbbRT server is processing requests at 2.75× the rate of Linux. This can be largely attributed to our shorter non-preemptive instruction path for processing requests. Observe that the Linux rate of instructions per request is 2.44× that of EbbRT. The instructions per cycle rate in EbbRT, a 12.6% increase over Linux, shows that we are running more efficiently overall. This can be again observed through our decreased per-cycle rates of last level cache (LLC) reference and icache misses, which, on Linux, increase by 1.21× and 4.27×, respectively.

The above efficiency results suggest that our performance advantages are largely achieved through the construction of specialized system software to take advantage of properties of the memcached workload. We illustrate this in greater detail by examining the per-request latency for EbbRT and Linux (native) broken down into time spent processing network ingress, application logic, and network egress. For Linux, we used the `perf` tool to gather stacktrace samples over 30 seconds of a fully loaded, single core memcached instance and categorized each trace. For EbbRT, we instrumented the source code with timestamp counters. Table 3 presents this result. It should be noted that, for Linux, the "Application" category includes time spent scheduling, context switching, and handling event notification (e.g. `epoll`). The latency breakdown demonstrates that the performance advantage comes from specialization across the entire software stack, and not just one component.

By writing to our interfaces, memcached is implemented to directly handle memory filled by the device, and can likewise send replies without copying. A request is handled synchronously from the device driver without pre-emption, which enables a significant performance advantage. EbbRT primitives, such as `IOBufs` and RCU data structures, are used throughout the ap-

	Ingress	Application	Egress	Total
EbbRT	0.89 µs	0.86 µs	0.83 µs	2.59 µs
Linux	1.05 µs	1.30 µs	1.46 µs	3.81 µs

Table 3: Memcached Per-Request Latency

plication to simplify the development of the zero-copy, lock-free code.

In the past, significant effort has gone into improving the performance of memcached and similar key-value stores. However, many of these optimizations require client modifications [35, 43] or the use of custom hardware [26, 36]. By writing memcached as an EbbRT application, we are able to achieve significant performance improvements while maintaining compatibility with standard clients, protocols, and hardware.

5.3 Node.js

It is often the case that specialized systems can demonstrate high performance for a particular workload, such as packet processing, but fail to provide similar benefits to more full-featured applications. A key objective of EbbRT is to provide an efficient base set of primitives on top of which a broad set of applications can be constructed.

We evaluate node.js, a popular JavaScript execution environment for server-side applications. In comparison to memcached, node.js uses many more features of an operating system, including virtual memory mapping, file I/O, periodic timers, etc. Node.js links with several C/C++ libraries to provide its event-driven environment. In particular, the two libraries which involved the most effort to port were V8 [23], Google's JavaScript engine, and libuv [34], which abstracts OS functionality and callback based event-driven execution.

Porting V8 was relatively straightforward as EbbRT supports the C++ standard library, on which V8 depends. Additional OS functionality required such as clocks, timers, and virtual memory, are provided by the core Ebbs of the system. Porting libuv required significantly more effort, as there are over 100 functions of the libuv interface which require OS specific implementations. In the end, our approach enables the libuv callbacks to be invoked directly from a hardware interrupt, in the same way that our memcached implementation receives incoming requests.

The effort to port node.js was significantly simplified by exploiting EbbRT's model of function offloading. For example, the port included the construction of an application-specific `FileSystem` Ebb. Rather than implement a file system and hard disk driver within the EbbRT library OS, the Ebb calls are offloaded to a (hosted) representative running in a Linux process. Our default implementation of the `FileSystem` Ebb is naïve, sending messages and incurring round trip costs for every access, rather than caching data on local representatives. For evaluation purposes we use a modified version of the `FileSystem` Ebb which performs no communication and serves a single static node.js script as `stdin`. This implementation allows us to evaluate the following workloads (which perform no file access) without also involving a hosted library.

One key observation of the node.js port is the modest development effort required to get a large piece of software functional, and, more importantly, the ability to reuse many of the software mechanisms used in our memcached application. The port was largely completed by a single developer in two weeks. Concretely, node.js and its dependencies total over one million lines of code, the majority of which is the v8 JavaScript engine. We wrote about 3000 lines of new code in order to support node.js on EbbRT. A significant factor in simplifying the port is the fact that EbbRT is distributed with a custom toolchain. Rather than needing to modify the existing node.js build system, we specified EbbRT as a target and built it as we would any other cross compiled binary. This illustrates EbbRT's support for a broad class of software as well as the manner in which we reduce developer burden required to develop specialized systems.

5.3.1 V8 JavaScript Benchmark

To compare the performance of our port to that of Linux, we launch node.js running version 7 of the V8 JavaScript benchmark suite [22]. This collection of purely compute-bound benchmarks stresses the core performance of the V8 JavaScript engine. Figure 8 shows the benchmark scores. Scores are computed by inverting the running time of the benchmark and scaling it by the score of a reference implementation (higher is better). The overall score is the geometric mean of the 8 individual scores. The figure normalizes each score to the Linux result.

EbbRT outperforms Linux run within a VM on each benchmark, with a 5.1% improvement in overall score. Most prominently, EbbRT is able to attain a 30.3% improvement in the memory intensive `Splay` benchmark. As we've made no modification to the V8 software, just running it on EbbRT accounts for the improved performance.

We further investigate the sources of the performance advantage by running the Linux `perf` utility to measure several CPU efficiency metrics. Table 4 displays these results. Several interesting aspects of this table deserve highlighting. First, EbbRT has a slightly better IPC efficiency (3.76%), which can in part be attributed to its performance advantage. One reason for decreased effi-

	Inst/cycle	LLC ref/cycle	TLB miss/cycle	VM exit	Hypervisor time	Guest kernel time
EbbRT	2.48	0.0021	1.18e-5	5950	0.33%	N/A
Linux VM	2.39	0.0028	9.92e-5	66851	0.74%	1.08%

Table 4: V8 JavaScript Benchmark CPU-efficiency metrics

ciency of the Linux VM is simply having to execute more instructions, such as additional VM Exits and extraneous kernel functionality (e.g., scheduling). Second, the additional interactions with the hypervisor and kernel on Linux increase the working set size and cause a 33% increase in LLC accesses. Third, Linux suffers nearly 9× more TLB misses than EbbRT. We attribute our TLB efficiency to our use of large pages throughout the system.

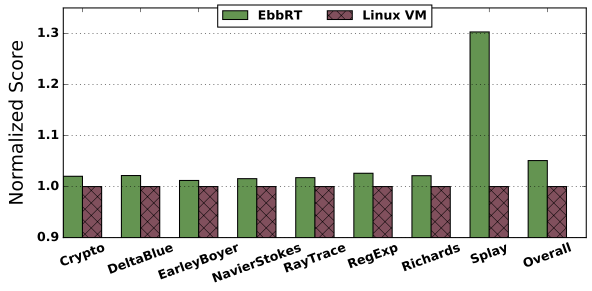


Figure 8: V8 JavaScript Benchmark

5.3.2 Node.js Webserver

Lastly, we evaluate a trivial webserver written for node.js, which uses the builtin `http` module and responds to each `GET` request with a small static message totaling 148 bytes. We use the `wrk` [20] benchmark to place moderate load on the webserver and measure mean and 99th percentile response-time latencies. EbbRT achieved 91.1 µs mean and 100.0 µs 99th percentile latencies. Linux achieved 103.5 µs mean and 120.6 µs 99th percentile latencies. The node.js webserver running on Linux has a 13.61% higher mean latency than the same webserver run on EbbRT. 99th percentile latency is 20.65% higher on Linux over EbbRT.

These results suggest that an entire class of server-side application written for node.js can achieve immediate performance advantages by simply running on top of EbbRT. Similar to our memcached evaluation, the ability for node.js to serve requests directly from hardware interrupts, without context switching or pre-emption, enables greater network performance. The non-preemptive run-to-completion execution model particularly improves tail latency. Our V8 benchmark results show that the use of large pages and simplified execution paths increases the efficiency of CPU and memory intensive workloads.

Finally, our approach opens up the application to further optimizations opportunities. For example, one could modify V8 to directly access the page tables to improve garbage collection [4]. We expect that greater performance can be achieved through continued system specialization.

6 Related Work

The Exokernel [13] introduced the library operating system structure — where system functionality is directly linked into the application and executes in the same address space and protection domain. Library operating systems have been shown to provide many useful properties, such as portability [45, 55], security [3, 38], and efficiency [28]. OSv [28] and Mirage [38] are similar to EbbRT in that they target virtual machines deployed in IaaS clouds. OSv constructs a general purpose library OS and supports the Linux ABI. Mirage uses the OCaml programming language to construct minimal systems for security. EbbRT takes a middle ground, supporting source-level portability for existing applications through rich C++ functionality and standard libraries, but avoiding general purpose OS interfaces.

CNK [42], Libra [1], Azul [53], and, more recently, Arrakis [44] and IX [5] have pursued architectures that enable specialized execution environments for performance sensitive data flow. While their approaches vary, these systems must each make a trade-off between targeting a narrow class of applications (e.g., HPC, Java web applications, or packet processing) and targeting a broad class of applications. Rather than supporting a single specialized execution environment, EbbRT provides a framework to enable the construction of various application-specific library operating systems.

Considerable work has been done on system software customization [8, 11, 17, 30, 50]. Much of this work focuses on developing general purpose operating that are customizable, while EbbRT is focused on the construction of specialized systems.

Choices [10] and OSKit [16] provide operating system frameworks; in the case of Choices, for maintainability and extensibility, and in the case of OSKit, to simplify the construction of new operating systems. EbbRT differs most significantly in its performance objectives and its focus on enabling application developers to extend and customize system software. For example, by providing EbbRT as a modified toolchain, application-level software libraries (e.g. boost) can be used in a systems context with little to no modification.

Others have considered the interaction of an object-oriented framework with the goal of enabling high performance. CHAOSarc [18] and TinyOS [32] have both explored the use of a fine grain object framework in the resource limited setting of embedded systems. Like TinyOS, EbbRT combines language support for event driven execution and method dispatch mechanisms that are friendly to compiler optimization. In a recent retrospective [33], the authors recognized that their development and use of the nesC programming language limited TinyOS from even broader, long-term use. EbbRT, however, focuses on integration with existing software tooling, development patterns, and libraries to encourage continued applicability.

7 Concluding Remarks

We have presented EbbRT, a framework for constructing specialized systems for cloud applications. Through our evaluation we have established that EbbRT applications achieve their performance advantages through system-wide specialization rather than one particular technique. In addition, we have shown that existing applications can be ported to EbbRT with modest effort and achieve a noticeable performance gain. Throughout this paper we have conveyed how our primary design elements, i.e., elastic building blocks, an event-driven execution environment, and a heterogeneous deployment model working alongside advanced language constructs and new system primitives, have proven to be a novel, effective approach to achieving our stated objectives.

By encapsulating system components with minimal dispatch overhead, we enable application-specific performance specialization throughout all parts of our system. Furthermore, we have shown that our default Ebb implementations provide a foundation for achieving performance advantages. For example, our results illustrate the combined benefits of using a non-preemptive event-driven execution model, identity mapped memory, and zero-copy paths.

As we gained experience with the system the issue of easing development efforts arose naturally. An early focus on enabling use of standard libraries, including the addition of blocking primitives, greatly simplified development. Our monadic futures implementation addressed concrete concerns we had with event-driven programming. Futures are now used throughout our implementation. IOBufs came about as a solution for us to enable pervasive zero-copy with little added complexity or overhead.

EbbRT's long-term utility hinges on its ability to be used for a broad range of applications, while continuing to enable a high degree of per-application specialization. Our previous work on fragmented objects gives us confidence that various specialized implementations of our existing Ebbs can be introduced, as needed, without diminishing the overall value and integrity of the framework.

Future work involves further exploration of system specialization. Our focus has primarily revolved around networked applications, however, data storage applications should equally benefit from specialization. Additionally, EbbRT can be used to accelerate existing applications in a fine-grained fashion. We believe the hosted library can be used not just for compatibility for new applications, but as a way to offload performance critical functionality to one or more library operating systems.

The EbbRT framework is open source and actively used in ongoing systems research. We invite developers and researchers alike to visit our online codebase at `https://github.com/sesa/ebbrt/`

Acknowledgments: We owe a great deal of gratitude to the Massachusetts Open Cloud (MOC) team for their help and support in getting EbbRT running and debugged on HIL [24]. We would like to thank Michael Stumm, Larry Rudolph and Frank Bellosa for feedback on early drafts of this paper. Thanks to Tommy Unger, Kyle Hogan and David Yeung for helping us get the nits out. We would also like to thank our shepherd, Andrew Baumann, for his help in preparing the final version. This work was supported by the National Science Foundation under award IDs CNS-1012798, CNS-1254029, CCF-1533663 and CNS-1414119. Early work was supported by by the Department of Energy under Award Number DE-SC0005365.

References

[1] Glenn Ammons, Jonathan Appavoo, Maria Butrico, Dilma Da Silva, David Grove, Kiyokuni Kawachiya, Orran Krieger, Bryan Rosenburg, Eric Van Hensbergen, and Robert W. Wisniewski. Libra: A Library Operating System for a Jvm in a Virtualized Execution Environment. In *Proceedings of the 3rd International Conference on Virtual Execution Environments*, VEE '07, pages 44–54. ACM, 2007.

[2] Berk Atikoglu, Yuehai Xu, Eitan Frachtenberg, Song Jiang, and Mike Paleczny. 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*, SIGMETRICS '12, pages 53–64. ACM, 2012.

[3] Andrew Baumann, Marcus Peinado, and Galen Hunt. Shielding Applications from an Untrusted Cloud with Haven. *ACM Trans. Comput. Syst.*, 33(3):8:1–8:26, August 2015.

[4] Adam Belay, Andrea Bittau, Ali Mashtizadeh, David Terei, David Mazières, and Christos Kozyrakis. Dune: Safe User-level Access to Privileged CPU Features. In *Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation*, OSDI'12, pages 335–348. USENIX Association, 2012.

[5] Adam Belay, George Prekas, Ana Klimovic, Samuel Grossman, Christos Kozyrakis, and Edouard Bugnion. IX: A Protected Dataplane Operating System for High Throughput and Low Latency. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*, pages 49–65, Broomfield, CO, October 2014. USENIX Association.

[6] Emery D. Berger, Kathryn S. McKinley, Robert D. Blumofe, and Paul R. Wilson. Hoard: A Scalable Memory Allocator for Multithreaded Applications. In *Proceedings of the Ninth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS IX, pages 117–128. ACM, 2000.

[7] Emery D. Berger, Benjamin G. Zorn, and Kathryn S. McKinley. Composing High-performance Memory Allocators. In *Proceedings of the ACM SIGPLAN 2001 Conference on Programming Language Design and Implementation*, PLDI '01, pages 114–124. ACM, 2001.

[8] Brian N. Bershad, Craig Chambers, Susan Eggers, Chris Maeda, Dylan McNamee, Przemysław Pardyak, Stefan Savage, and Emin Gün Sirer. SPIN - an Extensible Microkernel for Application-specific Operating System Services. *SIGOPS Oper. Syst. Rev.*, 29(1):74–77, January 1995.

[9] Georges Brun-Cottan and Mesaac Makpangou. Adaptable Replicated Objects in Distributed Environments. Research Report RR-2593, 1995. Project SOR.

[10] Roy H. Campbell, Nayeem Islam, and Peter Madany. Choices, frameworks and refinement. *Computing Systems*, 5(3):217–257, 1992.

[11] David R. Cheriton and Kenneth J. Duda. A Caching Model of Operating System Kernel Functionality. In *Proceedings of the 1st USENIX Conference on Operating Systems Design and Implementation*, OSDI '94. USENIX Association, 1994.

[12] Jonathan Corbet. SLQB - and then there were four. `http://lwn.net/Articles/311502`, Dec. 2008.

[13] D. R. Engler, M. F. Kaashoek, and J. O'Toole, Jr. Exokernel: An Operating System Architecture for Application-level Resource Management. In *Proceedings of the Fifteenth ACM Symposium on Operating Systems Principles*, SOSP '95, pages 251–266. ACM, 1995.

[14] Jason Evans. Scalable memory allocation using jemalloc. `http://www.canonware.com/jemalloc/`, 2011.

[15] Brad Fitzpatrick. Distributed Caching with Memcached. *Linux Journal*, 2004(124):5, August 2004.

[16] Bryan Ford, Godmar Back, Greg Benson, Jay Lepreau, Albert Lin, and Olin Shivers. The Flux OSKit: A Substrate for Kernel and Language Research. In *Proceedings of the Sixteenth ACM Symposium on Operating Systems Principles*, SOSP '97, pages 38–51. ACM, 1997.

[17] Ben Gamsa, Orran Krieger, Jonathan Appavoo, and Michael Stumm. Tornado: Maximizing Locality and Concurrency in a Shared Memory Multiprocessor Operating System. In *Proceedings of the Third Symposium on Operating Systems Design and Implementation*, OSDI '99, pages 87–100. USENIX Association, 1999.

[18] Ahmed Gheith and Karsten Schwan. CHAOSarc: Kernel Support for Multiweight Objects, Invocations, and Atomicity in Real-time Multiprocessor Applications. *ACM Trans. Comput. Syst.*, 11(1):33–72, February 1993.

[19] Sanjay Ghemawat and Paul Menage. Tcmalloc: Thread-caching malloc. `http://goog-perftools.sourceforge.net/doc/tcmalloc.html`, 2009.

[20] Will Glozer. wrk: Modern HTTP benchmarking tool. `https://github.com/wg/wrk`, 2014.

[21] Google. Protocol Buffers: Google's Data Interchange Format. `https://developers.google.com/protocol-buffers`.

[22] Google. V8 Benchmark Suit - Version 7. `https://v8.googlecode.com/svn/data/benchmarks/v7/`.

[23] Google. V8 JavaScript Engine. `http://code.google.com/p/v8/`.

[24] Jason Hennessey, Sahil Tikale, Ata Turk, Emine Ugur Kaynar, Chris Hill, Peter Desnoyers, and Orran Krieger. HIL: Designing an Exokernel for the Data Center. In *Proceedings of the Seventh ACM Symposium on Cloud Computing*, pages 155–168. ACM, 2016.

[25] Intel Corporation. Intel DPDK: Data Plane Development Kit. `http://dpdk.org`.

[26] Jithin Jose, Hari Subramoni, Miao Luo, Minjia Zhang, Jian Huang, Md. Wasi-ur Rahman, Nusrat S. Islam, Xiangyong Ouyang, Hao Wang, Sayantan Sur, and Dhabaleswar K. Panda. Memcached Design on High Performance RDMA Capable Interconnects. In *Proceedings of the 2011 International Conference on Parallel Processing*, ICPP '11, pages 743–752. IEEE Computer Society, 2011.

[27] Rishi Kapoor, George Porter, Malveeka Tewari, Geoffrey M. Voelker, and Amin Vahdat. Chronos: Predictable Low Latency for Data Center Applications. In *Proceedings of the Third ACM Symposium on Cloud Computing*, SoCC '12, pages 9:1–9:14. ACM, 2012.

[28] Avi Kivity, Dor Laor, Glauber Costa, Pekka Enberg, Nadav Har'El, Don Marti, and Vlad Zolotarov. OSv—Optimizing the Operating System for Virtual Machines. In *2014 USENIX Annual Technical Conference (USENIX ATC 14)*, pages 61–72. USENIX Association, June 2014.

[29] Kohlhoff, Christopher. Boost.Asio. `http://www.boost.org/doc/libs/1_55_0/doc/html/boost_asio.html`.

[30] Orran Krieger, Marc Auslander, Bryan Rosenburg, Robert W. Wisniewski, Jimi Xenidis, Dilma Da Silva, Michal Ostrowski, Jonathan Appavoo, Maria Butrico, Mark Mergen, Amos Waterland, and Volkmar Uhlig. K42: Building a Complete Operating System. In *Proceedings of the 1st ACM SIGOPS/EuroSys European Conference on Computer Systems 2006*, EuroSys '06, pages 133–145. ACM, 2006.

[31] Jacob Leverich. Mutilate: High-Performance Memcached Load Generator. `https://github.com/leverich/mutilate`, 2014.

[32] P. Levis, S. Madden, J. Polastre, R. Szewczyk, K. Whitehouse, A. Woo, D. Gay, J. Hill, M. Welsh, E. Brewer, and D. Culler. *TinyOS: An Operating System for Sensor Networks*, pages 115–148. Ambient Intelligence. Springer Berlin Heidelberg, 2005.

[33] Philip Levis. Experiences from a Decade of TinyOS Development. In *Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation*, OSDI'12, pages 207–220. USENIX Association, 2012.

[34] libuv. `http://libuv.org`.

[35] Hyeontaek Lim, Dongsu Han, David G. Andersen, and Michael Kaminsky. MICA: A Holistic Approach to Fast In-Memory Key-Value Storage. In *11th USENIX Symposium on Networked Systems Design and Implementation (NSDI 14)*, pages 429–444. USENIX Association, April 2014.

[36] Kevin Lim, David Meisner, Ali G. Saidi, Parthasarathy Ranganathan, and Thomas F. Wenisch. Thin Servers with Smart Pipes: Designing SoC Accelerators for Memcached. In *Proceedings of the 40th Annual International Symposium on Computer Architecture*, ISCA '13, pages 36–47. ACM, 2013.

[37] Barbara Liskov and Liuba Shrira. Promises: Linguistic Support for Efficient Asynchronous Procedure Calls in Distributed Systems. In *Proceedings of the ACM SIGPLAN 1988 Conference on Programming Language Design and Implementation*, PLDI '88, pages 260–267. ACM, 1988.

[38] Anil Madhavapeddy, Richard Mortier, Charalampos Rotsos, David Scott, Balraj Singh, Thomas Gazagnaire, Steven Smith, Steven Hand, and Jon Crowcroft. Unikernels: Library Operating Systems for the Cloud. *SIGPLAN Not.*, 48(4):461–472, March 2013.

[39] Mesaac Makpangou, Yvon Gourhant, and Jean pierre Le Narzul. Fragmented Objects for Distributed Abstractions. In *Readings in Distributed Computing Systems*, pages 170–186. IEEE Computer Society Press, 1992.

[40] Paul E. McKenney, Jonathan Appavoo, Andi Kleen, Orran Krieger, Rusty Russell, Dipankar Sarma, and Maneesh Soni. Read-Copy Update. In *Ottawa Linux Symposium*, July 2001.

[41] Greg Minshall, Yasushi Saito, Jeffrey C. Mogul, and Ben Verghese. Application Performance Pitfalls and TCP's Nagle Algorithm. *SIGMETRICS Perform. Eval. Rev.*, 27(4):36–44, March 2000.

[42] José Moreira, Michael Brutman, José Castaños, Thomas Engelsiepen, Mark Giampapa, Tom Gooding, Roger Haskin, Todd Inglett, Derek Lieber, Pat McCarthy, Mike Mundy, Jeff Parker, and Brian Wallenfelt. Designing a Highly-Scalable Operating System: The Blue Gene/L story. In *Proceedings of the 2006 ACM/IEEE conference on Supercomputing*, SC '06. ACM, 2006.

[43] Rajesh Nishtala, Hans Fugal, Steven Grimm, Marc Kwiatkowski, Herman Lee, Harry C. Li, Ryan McElroy, Mike Paleczny, Daniel Peek, Paul Saab, David Stafford, Tony Tung, and Venkateshwaran Venkataramani. Scaling Memcache at Facebook. In *Proceedings of the 10th USENIX Conference on Networked Systems Design and Implementation*, NSDI'13, pages 385–398. USENIX Association, 2013.

[44] Simon Peter, Jialin Li, Irene Zhang, Dan R. K. Ports, Doug Woos, Arvind Krishnamurthy, Thomas Anderson, and Timothy Roscoe. Arrakis: The Operating System is the Control Plane. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*, pages 1–16. USENIX Association, October 2014.

[45] Donald E. Porter, Silas Boyd-Wickizer, Jon Howell, Reuben Olinsky, and Galen C. Hunt. Rethinking the Library OS from the Top Down. In *Proceedings of the Sixteenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XVI, pages 291–304. ACM, 2011.

[46] Niels Provos and Nick Mathewson. libevent - an event notification library. `http://libevent.org/`, 2003.

[47] Injong Rhee, Nallathambi Balaguru, and George N. Rouskas. MTCP: Scalable TCP-like Congestion Control for Reliable Multicast. *Comput. Netw.*, 38(5):553–575, April 2002.

[48] Luigi Rizzo. Netmap: A Novel Framework for Fast Packet I/O. In *Proceedings of the 2012 USENIX Conference on Annual Technical Conference*, USENIX ATC'12, page 9. USENIX Association, 2012.

[49] Dan Schatzberg, James Cadden, Orran Krieger, and Jonathan Appavoo. A Way Forward: Enabling Operating System Innovation in the Cloud. In *6th USENIX Workshop on Hot Topics in Cloud Computing (HotCloud 14)*. USENIX Association, June 2014.

[50] Margo I. Seltzer, Yasuhiro Endo, Christopher Small, and Keith A. Smith. Dealing with disaster: Surviving misbehaved kernel extensions. In *Proceedings of the Second USENIX Symposium on Operating Systems Design and Implementation*, OSDI '96, pages 213–227, New York, NY, USA, 1996. ACM.

[51] Marc Shapiro, Yvon Gourhant, Sabine Habert, Laurence Mosseri, Michel Ruffin, and Céline Valot. SOS: An Object-Oriented Operating System - Assessment and Perspectives. *Computing Systems*, 2:287–337, 1991.

[52] Quinn O Snell, Armin R Mikler, and John L Gustafson. Netpipe: A Network Protocol Independent Performance Evaluator. In *IASTED International Conference on Intelligent Information Management and Systems*, 1996.

[53] Gil Tene, Balaji Iyengar, and Michael Wolf. C4: The Continuously Concurrent Compacting Collector. In *Proceedings of the International Symposium on Memory Management*, ISMM '11, pages 79–88. ACM, 2011.

[54] Ajay Tirumala, Feng Qin, Jon Dugan, Jim Ferguson, and Kevin Gibbs. Iperf: The TCP/UDP bandwidth measurement tool. `https://iperf.fr/`.

[55] Chia-Che Tsai, Kumar Saurabh Arora, Nehal Bandi, Bhushan Jain, William Jannen, Jitin John, Harry A. Kalodner, Vrushali Kulkarni, Daniela Oliveira, and Donald E. Porter. Cooperation

and Security Isolation of Library OSes for Multi-process Applications. In *Proceedings of the Ninth European Conference on Computer Systems*, EuroSys '14, pages 9:1–9:14. ACM, 2014.

[56] Maarten van Steen, Philip Homburg, and Andrew S. Tanenbaum. Globe: A Wide-Area Distributed System. *IEEE Concurrency*, 7(1):70–78, January 1999.

[57] S. Vinoski. CORBA: Integrating Diverse Applications Within Distributed Heterogeneous Environments. *Comm. Mag.*, 35(2):46–55, February 1997.

[58] Rob von Behren, Jeremy Condit, and Eric Brewer. Why Events Are a Bad Idea (for High-concurrency Servers). In *Proceedings of the 9th Conference on Hot Topics in Operating Systems - Volume 9*, HOTOS'03, page 4. USENIX Association, 2003.

[59] David A Wheeler. SLOCCount. `http://www.dwheeler.com/sloccount/`.

[60] Sara Williams and Charlie Kindel. The Component Object Model: A Technical Overview. *Dr. Dobbs Journal*, 356:356–375, 1994.

SCONE: Secure Linux Containers with Intel SGX

Sergei Arnautov[1], Bohdan Trach[1], Franz Gregor[1], Thomas Knauth[1], Andre Martin[1], Christian Priebe[2], Joshua Lind[2], Divya Muthukumaran[2], Dan O'Keeffe[2], Mark L Stillwell[2], David Goltzsche[3], David Eyers[4], Rüdiger Kapitza[3], Peter Pietzuch[2], and Christof Fetzer[1]

[1]*Fakultät Informatik, TU Dresden,* christof.fetzer@tu-dresden.de
[2]*Dept. of Computing, Imperial College London,* prp@imperial.ac.uk
[3]*Informatik, TU Braunschweig,* rrkapitz@ibr.cs.tu-bs.de
[4]*Dept. of Computer Science, University of Otago,* dme@cs.otago.ac.nz

Abstract

In multi-tenant environments, Linux containers managed by Docker or Kubernetes have a lower resource footprint, faster startup times, and higher I/O performance compared to virtual machines (VMs) on hypervisors. Yet their weaker isolation guarantees, enforced through software kernel mechanisms, make it easier for attackers to compromise the confidentiality and integrity of application data within containers.

We describe SCONE, a secure container mechanism for Docker that uses the SGX trusted execution support of Intel CPUs to protect container processes from outside attacks. The design of SCONE leads to (i) a small trusted computing base (TCB) and (ii) a low performance overhead: SCONE offers a secure C standard library interface that transparently encrypts/decrypts I/O data; to reduce the performance impact of thread synchronization and system calls within SGX enclaves, SCONE supports user-level threading and asynchronous system calls. Our evaluation shows that it protects unmodified applications with SGX, achieving 0.6×–1.2× of native throughput.

1 Introduction

Container-based virtualization [53] has become popular recently. Many multi-tenant environments use Linux containers [24] for performance isolation of applications, Docker [42] for the packaging of the containers, and Docker Swarm [56] or Kubernetes [35] for their deployment. Despite improved support for hardware virtualization [21, 1, 60], containers retain a performance advantage over *virtual machines* (VMs) on hypervisors: not only are their startup times faster but also their I/O throughput and latency are superior [22]. Arguably they offer weaker security properties than VMs because the host OS kernel must protect a larger interface, and often uses only software mechanisms for isolation [8].

More fundamentally, existing container isolation mechanisms focus on protecting the environment from accesses by untrusted containers. Tenants, however, want to protect the confidentiality and integrity of their application data from accesses by unauthorized parties—not only from other containers but also from higher-privileged system software, such as the OS kernel and the hypervisor. Attackers typically target vulnerabilities in existing virtualized system software [17, 18, 19], or they compromise the credentials of privileged system administrators [65].

Until recently, there was no widely-available hardware mechanism for protecting user-level software from privileged system software. In 2015, Intel released the *Software Guard eXtensions* (SGX) [31] for their CPUs, which add support for secure *enclaves* [26]. An enclave shields application code and data from accesses by other software, including higher-privileged software. Memory pages belonging to an enclave reside in the *enclave page cache* (EPC), which cannot be accessed by code outside of the enclave. This makes SGX a promising candidate for protecting containers: the application process of a container can execute inside an enclave to ensure the confidentiality and integrity of the data.

The design of a secure container mechanism using SGX raises two challenges: (i) *minimizing* the size of the *trusted computing base* (TCB) inside an enclave while supporting existing applications in secure containers; and (ii) maintaining a *low performance overhead* for secure containers, given the restrictions of SGX.

Regarding the TCB size, prior work [6] has demonstrated that Windows applications can be executed in enclaves, but at the cost of a large TCB (millions of LOC), which includes system libraries and a library OS. Any vulnerability in the TCB may allow an attacker to access application data or compromise its integrity, which motivates us to keep a container's TCB size inside of the enclave small.

The performance overhead of enclaves comes from the fact that, since the OS kernel is untrusted, enclave code cannot execute system calls. An enclave thread must copy memory-based arguments and leave the enclave before a system call. These thread transitions are expensive because they involve saving and restoring the enclave's execution state. In addition, enclaves have lower memory performance because, after cache misses, cache lines must be decrypted when fetched from memory. Accesses to enclave pages outside of the EPC cause expensive page faults.

To maintain a small TCB for secure containers, we observe that containers typically execute network services such as Memcached [23], Apache [44], NGINX [47] and Redis [46], which require only a limited interface for system support: they communicate with the outside via network sockets or `stdin/stdout` streams, use isolated or ephemeral file systems, and do not access other I/O devices directly. To mitigate the overhead of secure containers, we note that enclave code can access memory outside of the enclave without a performance penalty. However, for applications with a high system call frequency, the overhead of leaving and re-entering the enclave for each system call remains expensive.

We describe **SCONE**, a **Secure CONtainer Environment** for Docker that uses SGX to run Linux applications in secure containers. It has several desirable properties:

(1) Secure containers have a small TCB. SCONE exposes a *C standard library interface* to container processes, which is implemented by statically linking against a libc library [38] within the enclave. System calls are executed outside of the enclave, but they are *shielded* by transparently encrypting/decrypting application data on a per-file-descriptor basis: files stored outside of the enclave are therefore encrypted, and network communication is protected by transport layer security (TLS) [20]. SCONE also provides secure ephemeral file system semantics.

(2) Secure containers have a low overhead. To reduce costly enclave transitions of threads, SCONE provides a *user-level threading* implementation that maximizes the time that threads spend inside the enclave. SCONE maps OS threads to logical application threads in the enclave, scheduling OS threads between application threads when they are blocked due to thread synchronization.

SCONE combines this with an *asynchronous system call mechanism* in which OS threads outside the enclave execute system calls, thus avoiding the need for enclave threads to exit the enclave. In addition, SCONE reduces expensive memory accesses within the enclave by maintaining encrypted application data, such as cached files and network buffers, in non-enclave memory.

(3) Secure containers are transparent to Docker. Secure containers behave like regular containers in the Docker engine. Since container images are typically generated by experts, less experienced users can therefore benefit from SCONE, as long as they trust the creator of a secure container image. When executing secure containers, SCONE requires only an SGX-capable Intel CPU, an SGX kernel driver and an optional kernel module for asynchronous system call support.

Our experimental evaluation of SCONE on SGX hardware demonstrates that, despite the performance limitations of current SGX implementations, the throughput of popular services such as Apache, Redis, NGINX, and Memcached is 0.6×–1.2× of native execution, with a 0.6×–2× increase in code size. The performance of SCONE benefits from the asynchronous system calls and the transparent TLS encryption of client connections.

2 Secure Containers

Our goal is to create a *secure container* mechanism that protects the confidentiality and integrity of a Linux process' memory, code, and external file and network I/O from unauthorized and potentially privileged attackers.

2.1 Linux containers

Containers use OS-level virtualization [35] and have become increasingly popular for packaging, deploying and managing services such as key/value stores [46, 23] and web servers [47, 25]. Unlike VMs, they do not require hypervisors or a dedicated OS kernel. Instead, they use kernel features to isolate processes, and thus do not need to trap system calls or emulate hardware devices. This means that container processes can run as normal system processes, though features such as overlay file systems [10] can add performance overheads [22]. Another advantage of containers is that they are lightweight—they do not include the rich functionality of a standalone OS, but instead use the host OS for I/O operations, resource management, etc.

Projects such as LXC [24] and Docker [42] create containers using a number of Linux kernel features, including *namespaces* and the *cgroups* interface. By using the namespace feature, a parent process can create a child that has a restricted view of resources, including a remapped root file system and virtual network devices. The cgroups interface provides performance isolation between containers using scheduler features already present in the kernel.

For the deployment and orchestration of containers, frameworks such as Docker Swarm [56] and Kubernetes [35] instantiate and coordinate the interactions of containers across a cluster. For example, *micro-service* architectures [58] are built in this manner: a number of lightweight containers that interact over well-defined network interfaces.

2.2 Threat model

Analogous to prior work [50, 6], we assume a powerful and active adversary who has *superuser* access to the system and also access to the physical hardware. They can control the entire software stack, including privileged code, such as the container engine, the OS kernel, and other system software. This empowers the adversary to replay, record, modify, and drop any network packets or file system accesses.

We assume that container services were not designed with the above privileged attacker model in mind. They may compromise data confidentiality or integrity by trusting OS functionality. Any programming bugs or inadvertent design flaws in the application beyond trusting the OS are outside of our threat model, as mitigation would require orthogonal solutions for software reliability. In our threat model, we also do not target denial-of-service attacks, or side-channel attacks that exploit timing and page faults [63]. These are difficult to exploit in practice, and existing mitigation strategies introduce a high performance overhead [9].

2.3 Intel SGX

Intel's *Software Guard Extensions* (SGX) [29, 30, 15] allow applications to ensure confidentiality and integrity, even if the OS, hypervisor or BIOS are compromised. They also protect against attackers with physical access, assuming the CPU package is not breached.

Enclaves are *trusted execution environments* provided by SGX to applications. Enclave code and data reside in a region of protected physical memory called the *enclave page cache* (EPC). While cache-resident, enclave code and data are guarded by CPU access controls. When moved to DRAM, data in EPC pages is protected at the granularity of cache lines. An on-chip *memory encryption engine* (MEE) encrypts and decrypts cache lines in the EPC written to and fetched from DRAM. Enclave memory is also integrity protected meaning that memory modifications and rollbacks are detected.

Non-enclave code cannot access enclave memory, but enclave code can access untrusted DRAM outside the EPC directly, e.g., to pass function call parameters and results. It is the responsibility of the enclave code, however, to verify the integrity of all untrusted data.

Enclave life-cycle. Enclaves are created by untrusted code using the `ECREATE` instruction, which initializes an *SGX enclave control structure* (SECS) in the EPC. The `EADD` instruction adds pages to the enclave. SGX records the enclave to which the page was added, its virtual address and its permissions, and it subsequently enforces security restrictions, such as ensuring the enclave maps the page at the accessed virtual address. When all enclave pages are loaded, the `EINIT` instruction creates a cryptographic measurement, which can be used by remote parties for attestation.

For Intel Skylake CPUs [31], the EPC size is between 64 MB and 128 MB. To support enclave applications with more memory, SGX provides a paging mechanism for swapping pages between the EPC and untrusted DRAM: the system software uses privileged instructions to cause the hardware to copy a page into an encrypted buffer in DRAM outside of the EPC. Before reusing the freed EPC page, the system software must follow a hardware-enforced protocol to flush TLB entries.

Threading. After enclave initialization, an unprivileged application can execute enclave code through the `EENTER` instruction, which switches the CPU to enclave mode and jumps to a predefined enclave offset. Conversely, the `EEXIT` instruction causes a thread to leave the enclave. SGX supports multi-threaded execution inside enclaves, with each thread's enclave execution state stored in a 4 KB *thread control structure* (TCS).

Performance overhead. SGX incurs a performance overhead when executing enclave code: (i) since privileged instructions cannot execute inside the enclave, threads must exit the enclave prior to system calls. Such enclave transitions come at a cost—for security reasons, a series of checks and updates must be performed, including a TLB flush. Memory-based enclave arguments must also be copied between trusted and untrusted memory; (ii) enclave code also pays a penalty for writes to memory and cache misses because the MEE must encrypt and decrypt cache lines; and (iii) applications whose memory requirements exceed the EPC size must swap pages between the EPC and unprotected DRAM. Eviction of EPC pages is costly because they must be encrypted and integrity-protected before being copied to outside DRAM. To prevent address translation attacks, the eviction protocol interrupts all enclave threads and flushes the TLB.

2.4 Design trade-offs

Designing a secure Linux container using SGX requires a fundamental decision: *what system support should be placed inside an enclave to enable the secure execution of Linux processes in a container?* As we explore in this section, this design decision affects both (i) the *security properties* of containers, in terms of the size of the TCB and the exposed interface to the outside world, and (ii) the *performance* impact due to the inherent restric-

Service	TCB size	No. host system calls	Avg. throughput	Latency	CPU utilization
Redis	6.9×	<0.1×	0.6×	2.6×	1.1×
NGINX	5.7×	0.3×	0.8×	4.5×	1.5×
SQLite	3.8×	3.1×	0.3×	4.2×	1.1×

Table 1: Relative comparison of the LKL Linux library OS (no SGX) against native processes that use glibc

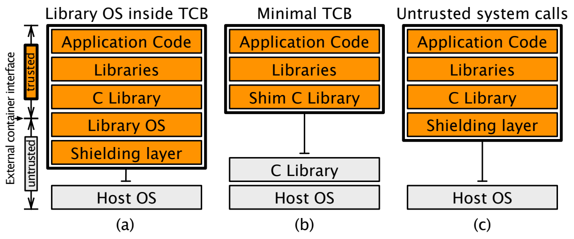


Figure 1: Alternative secure container designs

tions of SGX. To justify the design of SCONE, we first explore alternate design choices.

(1) External container interface. To execute unmodified processes inside secure containers, the container must support a C standard library (libc) interface. Since any libc implementation must use system calls, which cannot be executed inside of an enclave, a secure container must also expose an *external interface* to the host OS. As the host OS is untrusted, the external interface becomes an attack vector, and thus its design has security implications: an attacker who controls the host OS can use this interface to compromise processes running inside a secure container. A crucial decision becomes the size of (a) the external interface, and (b) the TCB required to implement the interface within the enclave.

Figure 1a shows a prior design point, as demonstrated by *Haven* [6], which minimizes the external interface by placing an entire Windows library OS inside the enclave. A benefit of this approach is that it exposes only a small external interface with 22 calls because a large portion of a process' system support can be provided by the library OS. The library OS, however, increases the TCB size inside of the enclave. In addition, it may add a performance overhead due to the extra abstractions (e.g., when performing I/O) introduced by the library OS.

We explore a similar design for Linux container processes. We deploy three typical containerized services using the *Linux Kernel Library* (LKL) [45] and the *musl* libc library [38], thus building a simple Linux library OS. The external interface of LKL has 28 calls, which is comparable to Haven.

Table 1 reports the performance and resource metrics for each service using the Linux library OS compared to a native *glibc* deployment. On average, the library OS increases the TCB size by 5×, the service latency by 4× and halves the service throughput. For Redis and NGINX, the number of system calls that propagate to the untrusted host OS are reduced as the library OS can handle many system calls directly. For SQLite, however, the number of system calls made to the host OS increases because LKL performs I/O at a finer granularity.

While our library OS lacks optimizations, e.g., minimizing the interactions between the library OS and the host OS, the results show that there is a performance degradation for both throughput and latency due to the kernel abstractions of the library OS. We conclude that the large TCB inside of the enclave and the performance overhead of this design is not a natural fit for containers.

Figure 1b shows the opposite, extreme design point: the external interface is used to perform all libc library calls made by the application. This raises the challenge of protecting the confidentiality and integrity of application data whilst exposing a wide interface. For example, I/O calls such as `read` and `write` could be used to compromise data within the enclave, and code inside the secure container cannot trust returned data. A benefit of this approach is that it leads to a minimal TCB inside the enclave—only a small shim C library needs to relay libc calls to the host libc library outside of the enclave.

Finally, Figure 1c shows a middle ground by defining the external interface at the level of system calls executed by the libc implementation. As we describe in §3, the design of SCONE explores the security and performance characteristics of this particular point in the design space. Defining the external container interface around system calls has the advantage that system calls already implement a privileged interface. While this design does not rely on a minimalist external interface to the host OS, we show that shield libraries can be used to protect a security-sensitive set of system calls: file descriptor based I/O calls, such as `read`, `write`, `send`, and `recv`, are shielded by transparently encrypting and decrypting the user data. While SCONE does not support some system calls, such as `fork`, `exec`, and `clone`, due to its user space threading model and the architectural limitations of SGX, they were not essential for the microservices that we targeted.

(2) System call overhead. All designs explored above pay the cost of executing system calls outside of the enclave (see §2.3). For container services with a high system call frequency, e.g., network-heavy services, this may result in a substantial performance impact. To quantify this issue, we conduct a micro-benchmark on an Intel Xeon CPU E3-1230 v5 at 3.4 GHz measuring the maximum rate at which `pwrite` system calls can be executed with and without an enclave. The benchmark is implemented using the Intel SGX SDK for Linux [32], which

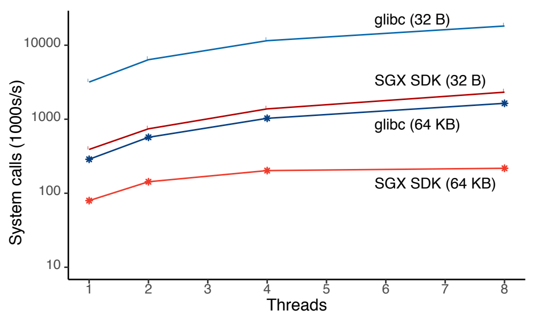


Figure 2: Number of executed `pwrite` system calls with an increasing number of threads

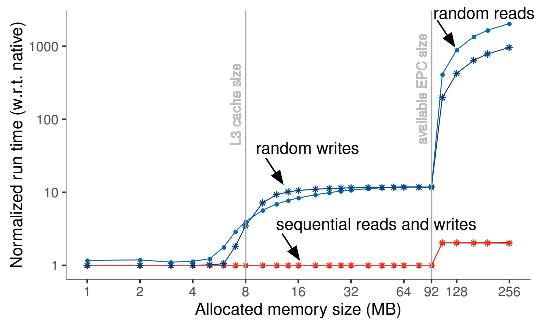


Figure 3: Normalized overhead of memory accesses with SGX enclaves

performs synchronous system calls with threads leaving and re-entering the enclave. We vary the number of threads and the `pwrite` buffer size.

Figure 2 shows that the enclave adds an overhead of an order of magnitude. The performance with large buffer sizes is limited by the copy overhead of the memory-based arguments; with small buffers, the main cost comes from the threads performing enclave transitions. We conclude that efficient system call support is a crucial requirement for secure containers. A secure container design must therefore go beyond simple synchronous support for system calls implemented using thread transitions.

(3) Memory access overhead. The memory accesses of a secure container process are affected by the higher overhead of accessing enclave pages (see §2.3). We explore this overhead using a micro-benchmark built with the Linux SGX SDK on the same hardware. The benchmark measures the time for both sequential and random read/write operations, normalized against a deployment without an enclave. All operations process a total of 256 MB, but access differently-sized memory regions.

Figure 3 shows that, as long as the accessed memory fits into the 8 MB L3 cache, the overheads are negligible. With L3 cache misses, there is a performance overhead of up to 12× for the random memory accesses. When the accessed memory is beyond the available EPC size, the triggered page faults lead to an overhead of three orders of magnitude. Sequential operations achieve better performance due to CPU prefetching, which hides some of the decryption overheads: they experience no overhead for memory ranges within the EPC size and a 2× overhead for sizes beyond that.

These results show that, for performance reasons, a secure container design should reduce access to enclave memory. Ideally, it should use untrusted non-enclave memory as much as possible, without compromising the offered security guarantees.

3 SCONE Design

Our objective is to offer *secure containers* on top of an untrusted OS: a secure container must protect containerized services from the threats defined in §2.2. We also want secure containers to fit transparently into existing Docker container environments: system administrators should be able to build secure container images with the help of Docker in a trusted environment and run secure containers in an untrusted environment.

3.1 Architecture

Figure 4 gives an overview of the SCONE architecture:

(1) SCONE exposes an *external interface* based on system calls to the host OS, which is shielded from attacks. Similar to what is done by the OS kernel to protect itself from user space attacks, SCONE performs sanity checks and copies all memory-based return values to the inside of the enclave before passing the arguments to the application (see §3.4). To protect the integrity and confidentiality of data processed via file descriptors, SCONE supports transparent encryption and authentication of data through *shields* (see §3.2).

(2) SCONE implements *M:N threading* to avoid the cost of unnecessary enclave transitions: M enclave-bound application threads are multiplexed across N OS threads. When an application thread issues a system call, SCONE checks if there is another application thread that it can wake and execute until the result of the system call is available (see §3.3).

(3) SCONE offers container processes an *asynchronous system call interface* to the host OS. Its implementation uses shared memory to pass the system call arguments and return values, and to signal that a system call should be executed. System calls are executed by separate threads running in a SCONE kernel module. Hence, the threads inside the enclave do not have to exit when performing system calls (see §3.4).

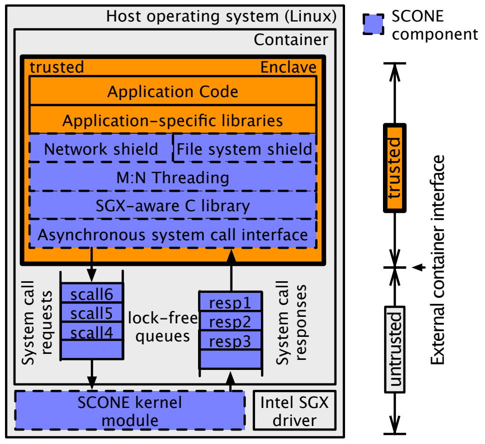


Figure 4: SCONE architecture

(4) SCONE integrates with existing Docker container environments, and ensures that secure containers are compatible with standard Linux containers (see §3.5). The host OS, however, must include a Linux SGX driver and, to boost performance, a SCONE kernel module. Note that SCONE does not use any functionality from the Intel Linux SDK [32] apart from the Linux SGX driver.

3.2 External interface shielding

So far, many popular services, such as Redis and Memcached, have been created under the assumption that the underlying OS is trusted. Such services therefore store files in the clear, communicate with other processes via unencrypted TCP channels (i.e., without TLS), and output to `stdout` and `stderr` directly.

To protect such services in secure containers, SCONE supports a set of *shields*. Shields focus on (1) preventing low-level attacks, such as the OS kernel controlling pointers and buffer sizes passed to the service (see §3.4); and (2) ensuring the confidentiality and integrity of the application data passed through the OS. A shield is enabled by statically linking the service with a given *shield library*. SCONE supports shields for (1) the transparent encryption of files, (2) the transparent encryption of communication channels via TLS, and (3) the transparent encryption of console streams.

When a file descriptor is opened, SCONE can associate the descriptor with a shield. A shield also has configuration parameters, which are encrypted and can be accessed only after the enclave has been initialized.

Note that the shields described below focus only on application data, and do not verify data maintained by the OS, such as file system metadata. If the integrity of such data is important, further shields can be added.

File system shield. The file system shield protects the confidentiality and integrity of files: files are authenticated and encrypted, transparently to the service. For the file system shield, a container image creator must define three disjoint sets of file path prefixes: prefixes of (1) *unprotected* files, (2) *encrypted and authenticated* files, and (3) *authenticated* files. When a file is opened, the shield determines the longest matching prefix for the file name. Depending on the match, the file is authenticated, encrypted, or just passed through to the host OS.

The file system shield splits files into blocks of fixed sizes. For each block, the shield keeps an authentication tag and a nonce in a metadata file. The metadata file is also authenticated to detect modifications. The keys used to encrypt and authenticate files as well as the three prefix sets are part of the configuration parameters passed to the file system shield during startup. For immutable file systems, the authentication tag of the metadata file is part of the configuration parameters for the file system shield. At runtime the metadata is maintained inside the enclave.

Containerized services often exclusively use a read-only file system and consider writes to be ephemeral. While processes in a secure container have access to the standard Docker `tmpfs`, it requires costly interaction with the kernel and its file system implementation. As a lightweight alternative, SCONE also supports a dedicated secure *ephemeral file system* through its file system shield. The shield ensures the integrity and confidentiality of ephemeral files: the ephemeral file system maintains the state of modified files in non-enclave memory. Our evaluation results show that the performance of ephemeral files is better than those of `tmpfs` (see §4.3).

The ephemeral file system implementation is resilient against *rollback attack*: after restarting the container process, the file system returns to a preconfigured startup state that is validated by the file system shield, and therefore it is not possible for an attacker to rollback the file system to an intermediate state. This is also true during runtime, since the metadata for files' blocks resides within the enclave.

Network shield. Some container services, such as Apache [44] and NGINX [47], always encrypt network traffic; others, such as Redis [46] and Memcached [23], assume that the traffic is protected by orthogonal means, such as TLS proxies, which terminate the encrypted connection and forward the traffic to the service in plaintext. Such a setup is appropriate only for data centers in which the communication between the proxy and the service is assumed to be trusted, which is incompatible with our threat model: an attacker could control the unprotected channel between the proxy and the service and modify the data. Therefore, for secure containers, a TLS network connection must be terminated inside the enclave.

SCONE permits clients to establish secure tunnels to

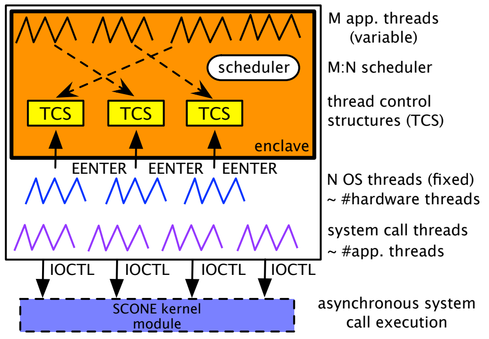

scheduler
TCS
TCS
TCS
enclave
EENTER
EENTER
EENTER
IOCTL
IOCTL
IOCTL
IOCTL
SCONE kernel module
M app. threads (variable)
M:N scheduler
thread control structures (TCS)
N OS threads (fixed) ~ #hardware threads
system call threads ~ #app. threads
asynchronous system call execution

Figure 5: M:N threading model

container services using TLS. It wraps all socket operations and redirects them to a *network shield*. The network shield, upon establishing a new connection, performs a TLS handshake and encrypts/decrypts any data transmitted through the socket. This approach does not require client- or service-side changes. The private key and certificate are read from the container's file system. Thus, they are protected by the file system shield.

Console shield. Container environments permit authorized processes to attach to the `stdin`, `stdout`, and `stderr` console streams. To ensure the confidentiality of application data sent to these streams, SCONE supports transparent encryption for them. The symmetric encryption key is exchanged between the secure container and the SCONE client during the startup procedure (see §3.5).

Console streams are unidirectional, which means that they cannot be protected by the network shield whose underlying TLS implementation requires bidirectional streams. A *console shield* encrypts a stream by splitting it into variable-sized blocks based on flushing patterns. A stream is protected against replay and reordering attacks by assigning each block a unique identifier, which is checked by the authorized SCONE client.

3.3 Threading model

SCONE supports an M:N threading model in which M application threads inside the enclave are mapped to N OS threads. SCONE thus has fewer enclave transitions, and, even though the maximum thread count must be specified at enclave creation time in SGX version 1 [29], SCONE supports a variable number of application threads.

As shown in Figure 5, multiple OS threads in SCONE can enter an enclave. Each thread executes the *scheduler*, which checks if: (i) an application thread needs to be woken due to an expired timeout or the arrival of a system call response; or (ii) an application thread is waiting to be scheduled. In both cases, the scheduler executes the associated thread. If no threads can be executed, the scheduler backs off: an OS thread may choose to sleep outside of the enclave when the back-off time is longer than the time that it takes to leave and reenter the enclave.

The number of OS threads inside the enclave is typically bound by the number of CPU cores. In this way, SCONE utilizes all cores without the need for a large number of OS threads inside the enclave. The scheduler does not support preemption. This is not a limitation in practice because almost all application threads perform either system calls or synchronization primitives at which point the scheduler can reschedule threads.

In addition to spawning N OS threads inside the enclave, SCONE also "captures" several OS threads inside the SCONE kernel module. The threads dequeue requests from the system call request queue, perform system calls, and enqueue results into the response queue (see Figure 4). The system call threads reside in the kernel indefinitely to eliminate the overhead of kernel mode switches. The number of system call threads must be at least the number of application threads to avoid stalling when system call threads block. Periodically, the system call threads leave the kernel module to trigger Linux housekeeping tasks, such as the cleanup of TCP state. When there are no pending system calls, the threads back-off exponentially to reduce CPU load.

SCONE does not support the `fork` system call. Enclave memory is tied to a specific process, and therefore the execution of `fork` would require the allocation, initialization, and attestation of an independent copy of an enclave. In current SGX implementations, the OS kernel cannot copy enclave memory to achieve this.

3.4 Asynchronous system calls

Since SGX does not allow system calls to be issued from within an enclave, they must be implemented with the help of calls to functions outside of the enclave. This means that the executing thread must copy memory-based arguments to non-enclave memory, exit the enclave and execute the outside function to issue the system call. When the system call returns, the thread must reenter the enclave, and copy memory-based results back to the enclave. As we showed in §2.4, such *synchronous system calls* have acceptable performance only for applications with a low system call rate.

To address this problem, SCONE also provides an *asynchronous system call interface* [52] (see Figure 6). This interface consists of two lock-free, multi-producer, multi-consumer queues: a *request queue* and a *response queue*. System calls are issued by placing a request into the request queue. An OS thread inside the SCONE kernel module receives and processes these requests. When the system call returns, the OS thread places the result into the response queue.

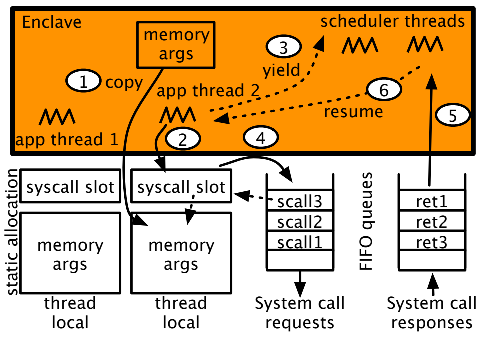


Figure 6: Asynchronous system calls

As shown in Figure 6, an application thread first copies memory-based arguments outside of the enclave ① and adds a description of the system call to a `syscall_slot` data structure ②, containing the system call number and arguments. The `syscall_slot` and the arguments use thread-local storage, which is reused by subsequent system calls. Next the application thread yields to the scheduler ③, which will execute other application threads until the reply to the system call is received in the response queue. The system call is issued by placing a reference to the `syscall slot` into the request queue ④. When the result is available in the response queue ⑤, buffers are copied to the inside of the enclave, and all pointers are updated to point to enclave memory buffers. As part of the copy operation, there are checks of the buffer sizes, ensuring that no *malicious* pointers referring to the outside of an enclave can reach the application. Finally, the associated application thread is scheduled again ⑥.

The enclave code handling system calls also ensures that pointers passed by the OS to the enclave do not point to enclave memory. This check protects the enclave from memory-based Iago attacks [12] and is performed for all shield libraries.

3.5 Docker integration

We chose to integrate SCONE with Docker because it is the most popular and widely used container platform. A future version of SCONE may use the open container platform [28], which would make it compatible with both Docker and rkt (CoreOS) [48]. With SCONE, a secure container consists of a single Linux process that is protected by an enclave, but otherwise it is indistinguishable from a regular Docker container, e.g., relying on the shared host OS kernel for the execution of system calls.

The integration of secure containers with Docker requires changes to the build process of secure images, and client-side extensions for spawning secure containers and for secure communication with these containers.

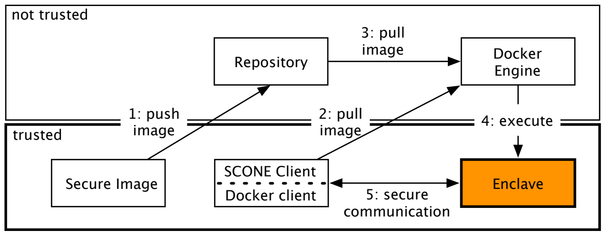


Figure 7: Using secure containers with Docker

SCONE does not require modifications to the Docker Engine or its API, but it relies on a wrapper around the original Docker client. A *secure SCONE client* is used to create configuration files and launch containers in an untrusted environment. SCONE supports a typical Docker workflow: a developer publishes an image with their application, and a user can customize the image by adding extra layers.

Image creation. Images are created in a trusted environment (see Figure 7). The image creator must be familiar with the security-relevant aspects of the service, e.g., which files to protect and which shields to activate.

To create a secure container image, the image creator first builds a SCONE executable of the application. They statically compile the application with its library dependencies and the SCONE library. SCONE does not support shared libraries by design to ensure that all enclave code is verified by SGX when an enclave is created.

Next, the image creator uses the SCONE client to create the metadata necessary to protect the file system. The client encrypts specified files and creates a *file system (FS) protection* file, which contains the message authentication codes (MACs) for file chunks and the keys used for encryption. The FS protection file itself is encrypted and added to the image. After that, the secure image is published using standard Docker mechanisms. SCONE does not need to trust the Docker registry, because the security-relevant parts are protected by the FS protection file.

If the image creator wants to support the composition of a secure Docker image [42], they only sign the FS protection file with their public key, but do not encrypt it. In this way, only its integrity is ensured, permitting additional customization. The confidentiality of the files is assured only after finishing the customization process.

Container startup. Each secure container requires a *startup configuration file* (SCF). The SCF contains keys to encrypt standard I/O streams, a hash of the FS protection file and its encryption key, application arguments and environment variables. Only an enclave whose identity has been verified can access the SCF. Since SGX does not protect the confidentiality of enclave code, embedding the startup configuration in the enclave itself is not an option. Instead, after the executable has initialized the enclave, the SCF is received through a TLS-protected network connection, established during enclave startup [2]. In production use, the container owner would validate that the container is configured securely before sending it the SCF. The SGX remote attestation mechanism [29] can attest to the enclave to enable this validation, but our current SCONE prototype does not support remote attestation.

4 Evaluation

Our evaluation of SCONE on SGX hardware is split into three parts: (i) we present application benchmarks for Apache [44], NGINX [47], Redis [46] and Memcached [23]. We compare the performance of these applications with SCONE against native variants (§4.2); (ii) we evaluate the performance impact of SCONE's file system shield with a set of micro-benchmarks (§4.3); and (iii) we discuss results from a micro-benchmark regarding the system call overhead (§4.4).

4.1 Methodology

All experiments use an Intel Xeon E3-1270 v5 CPU with 4 cores at 3.6 GHz and 8 hyper-threads (2 per core) and 8 MB cache. The server has 64 GB of memory and runs Ubuntu 14.04.4 LTS with Linux kernel version 4.2. We disable dynamic frequency scaling to reduce interference. The workload generators run on a machine with two 14-core Intel Xeon E5-2683 v3 CPUs at 2 GHz with 112 GB of RAM and Ubuntu 15.10. Each machine has a 10 Gb Ethernet NIC connected to a dedicated switch. The disk configuration is irrelevant as the workloads fit entirely into memory.

We evaluate two web servers, Apache [44], and NGINX [47]; Memcached [23]; Redis [46]; and SQLite [55]. The applications include a mix of compute (e.g., SQLite) and I/O intensive (e.g., Apache and Memcached) workloads. We compare the performance of three variants for each application: (i) one built with the GNU C library (glibc); (ii) one built with the musl [38] C library adapted to run inside SGX enclaves with synchronous system calls (SCONE-sync); and (iii) one built with the same musl C library but with asynchronous system calls (SCONE-async). We compare with glibc because it is the standard C library for most Linux distributions, and constitutes a more conservative baseline than musl. In our experiments, applications compiled against glibc perform the same or better than the musl-based variants. The application process (and Stunnel) execute inside a Docker container.

Application	Worker threads		Enclave threads		Syscall threads	
	async	sync	async	sync	async	sync
Apache	25	25	4	8	32	-
NGINX	1	1	1	1	16	-
Redis	1	1	1	1	16	-
Memcached	4	8	4	8	32	-

Table 2: Thread configuration used for applications

SCONE-async uses the SCONE kernel module to capture system call threads in the kernel. For each application and variant, we configure the number of threads (see §3.3) to give the best results, as determined experimentally. We summarize the thread configuration in Table 2. Worker threads are threads created by the application, e.g., using `pthread_create()`. In the glibc variant, worker threads are real OS threads, while in SCONE they represent user space threads. Enclave threads are OS threads that run permanently inside the enclave, while system call threads are OS threads that run permanently outside. With SCONE-sync, there are no dedicated system call threads because the enclave threads synchronously exit the enclave to perform system calls.

For applications that do not support encryption (e.g., Memcached and Redis), we use Stunnel [61] to encrypt their communication in the glibc variant. When reporting CPU utilization, the application's glibc variant includes the utilization due to Stunnel processes. In SCONE, the network shield subsumes the functionality of Stunnel.

Reported data points are based on ten runs, and we compute the 30% trimmed mean (i.e., without the top and bottom 30% outliers) and its variance. The trimmed mean is a robust estimator insensitive to outliers: it measures the central tendency even with jitter. Unless stated otherwise, the variance is small, and we omit error bars.

4.2 Application benchmarks

Apache is a highly configurable and mature web server, originally designed to spawn a process for each connection. This differs from the architecture of the other benchmarked web server—NGINX employs an event-driven design. By default, it uses a single thread but current versions can be configured to use multiple threads.

We use wrk2 [62] to fetch a web page. We increase the number of concurrent clients and the frequency at which they retrieve the page until the response times start to degrade. Since Apache supports application-level encryption in the form of HTTPS, we do not use Stunnel or SCONE's network shield.

Figure 8a shows that all three variants exhibit comparable performance until about 32,000 requests per sec-

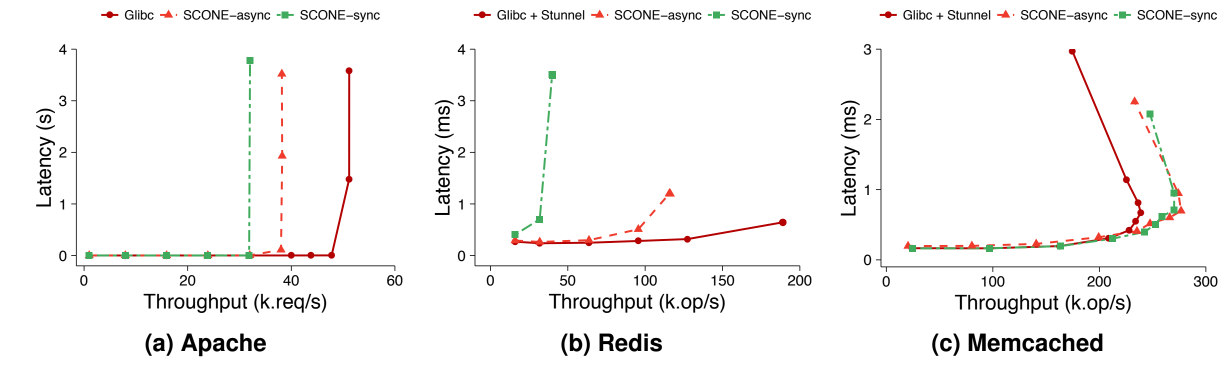


Figure 8: Throughput versus latency for Apache, Redis, and Memcached

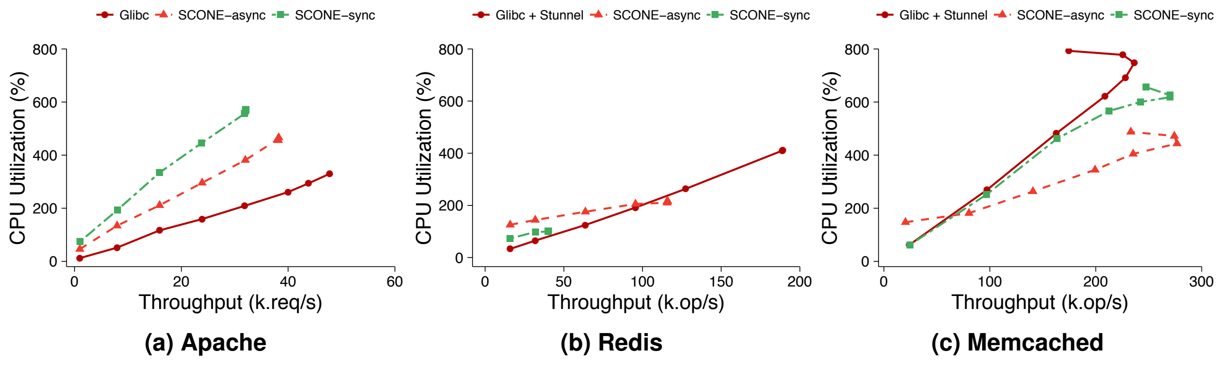


Figure 9: CPU utilization for Apache, Redis, and Memcached

ond, at which point the latency of the SCONE-sync increases dramatically. SCONE-async performs slightly better, reaching 38,000 requests per second. The glibc variant achieves 48,000 requests per second.

As shown in Figure 9a, SCONE-sync utilizes the CPU more despite the fact that SCONE-async uses extra threads to execute system calls. As we show below, the synchronous system call interface is not as performant as the asynchronous interface, resulting in a higher CPU utilization. However, SCONE-async has a higher CPU utilization than glibc. This is caused by the slower execution time of Apache running inside the enclave as well as the extra threads used in the SCONE kernel module to execute the system calls.

Redis is a distributed in-memory key/value store and represents an I/O-intensive network service. Typical workloads with many concurrent operations exhibit a high system call frequency. Persistence in Redis is achieved by forking and writing the state to stable storage in the background. Fundamentally, forking for enclave applications is difficult to implement and not supported by SCONE. Hence, we deploy Redis solely as an in-memory store.

We use workloads A to D from the YCSB benchmark suite [14]. In these workloads, both the application code and data fit into the EPC, so the SGX driver does not need to page-in EPC pages. We present results only for workload A (50% reads and 50% updates); the other workloads exhibit similar behaviour. We deploy 100 clients and increase their request frequency until reaching maximum throughput.

Figure 8b shows that Redis with glibc achieves a throughput of 189,000 operations per second. At this point, as shown in Figure 9b, Redis, which is single-threaded, becomes CPU-bound with an overall CPU utilization of 400% (4 hyper-threads): 1 hyper-thread is used by Redis, and 3 hyper-threads are used by Stunnel.

SCONE-sync cannot scale beyond 40,000 operations per second (21% of glibc), also due to Redis' single application thread. By design, SCONE-sync performs encryption as part of the network shield within the application thread. Hence, it cannot balance the encryption overhead across multiple hyper-threads, as Stunnel does, and its utilization peaks at 100%.

SCONE-async reaches a maximum throughput of 116,000 operations per second (61% of glibc). In addition to the single application thread, multiple OS threads execute system calls inside the SCONE kernel module. Ultimately, SCONE-async is also limited by the single Redis application thread, which is why CPU utilization peaks at 200% under maximum throughput. The per-

Variant	LOC (1000s)	libc.a Size (KB)	Apache Size (KB)	NGINX Size (KB)	Redis Size (KB)	Memcached Size (KB)	SQLite Size (KB)
glibc	1195	4710	5437	3975	2445	1143	801
musl	88	1498	4546	3088	682	289	832
SCONE libc	97	1572	4829	3286	853	357	880
Shielded SCONE libc	187	2491	5496	4082	1665	1208	1724

Table 3: Comparison of the binary sizes of statically linked applications using SCONE with native libc variants

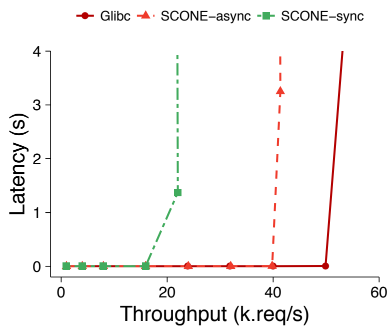


Figure 10: Throughput versus latency for NGINX

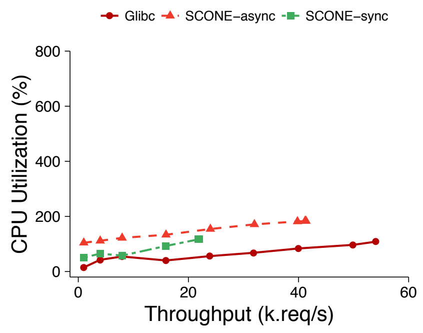


Figure 11: CPU utilization for NGINX

formance of SCONE-async is better than SCONE-sync because SCONE-async has a higher single thread system call throughput. However, as SCONE-async does not assign TLS termination to a separate thread either, it cannot reach the throughput of glibc.

Memcached, a popular key/value cache, is evaluated with the same YCSB workloads as Redis. We increase the number of clients until each variant reaches its saturation point. Again, the application fits into the EPC.

Figure 8c shows that the client latencies of all three variants exhibited for a given throughput are similar until approximately 230,000 operations per second, at which point the latency of glibc starts to increase faster than that of SCONE-async and SCONE-sync. The maximum achieved throughput of the SCONE variants (277,000 operations per second for SCONE-async and 270,000 operations per second for SCONE-sync) is higher than that of the glibc variant (238,000 operations per second).

For all three variants, the CPU utilization in Figure 9c increases with throughput. Both SCONE variants experience a lower CPU utilization than the Memcached and Stunnel deployment. This differs from single-threaded Redis, as Memcached can utilize more CPU cores with multiple threads and starts to compete for CPU cycles with Stunnel. SCONE's network shield encryption is more efficient, allowing it to have a lower CPU load and achieve higher throughput.

NGINX is a web server with an alternative architecture to Apache—NGINX typically uses one worker process per CPU core. Each worker process executes a non-blocking, event-driven loop to handle connections, process requests and send replies. We configure NGINX to use a single worker process.

Figure 10 and Figure 11 show the throughput and CPU utilization for NGINX, respectively. The glibc variant achieves approximately 50,000 requests per second—similar to Apache, but at a much lower utilization (>300% vs. 100%). SCONE-sync shows good performance up to 18,000 requests per second. This is less than Apache, but NGINX also only utilizes a single thread. With SCONE-async, NGINX achieves 80% of the native performance again at a much lower overall CPU utilization than Apache (200% vs. 500%). This demonstrates that SCONE can achieve acceptable performance both for multi-threaded applications, such as Apache, and non-blocking, event-based servers, such as NGINX.

Code size. We compare the code sizes of our applications in Table 3. The size of applications linked against the musl C library is in most cases smaller than that of applications linked against glibc. The modifications to musl for running inside of enclaves add 11,000 LOC, primarily due to the asynchronous system call wrappers (5000 LOC of generated code). The shields increase the code size further (around 99,000 LOC), primarily due to the TLS library. Nevertheless, the binary sizes of shielded applications increase only to 0.6×–2× compared to the glibc-linked variants.

Although the binary size of the SCONE libc variant is only 74 KB larger than musl, the binary size of some programs compiled against SCONE increase by more than this amount. This is due to how we build the SCONE

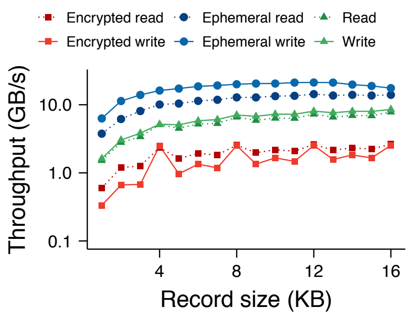


Figure 12: Throughput of random reads/writes with ephemeral file system versus `tmpfs`

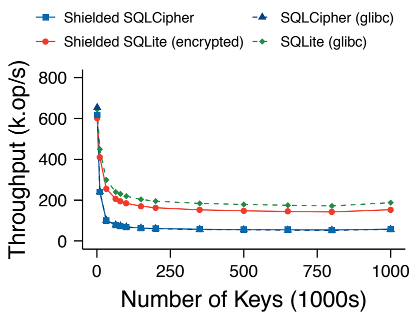


Figure 13: Throughput of SQLite and SQLCipher with file system shield

Application	SCONE-async	
	T'put	CPU util.
Apache	0.8×	1.4×
Redis	0.6×	1.0×
Memcached	1.2×	0.6×
NGINX	0.8×	1.8×

Table 4: Normalized application performance

binary: an application and all its dependent libraries are linked into a position-independent shared object file. The relocation information included in the shared object file comprises up to a few hundred kilobytes.

Discussion. Table 4 summarizes the normalized results for all the throughput-oriented applications. For Apache, SCONE-async achieves performance on par with that of the native version. The performance of SCONE-async is, as expected, faster than that of SCONE-sync—SCONE-async can switch to another Apache thread while waiting for system call results.

For single-threaded applications such as Redis, asynchronous system calls offer limited benefit, mostly due to the faster response times compared to synchronous system calls. However, SCONE-async cannot run other application threads while waiting for the result of a system call. The same is true for NGINX despite supporting multiple threads. In our experiments, NGINX did not scale as well as Apache with the same number of threads.

The results demonstrate that SCONE-async can execute scalable container services with throughput and latency that are comparable to native versions. This is in some sense surprising given the micro-benchmarks from §2.4, as they would suggest that applications inside SGX enclaves would suffer a more serious performance hit.

4.3 File system shield

We evaluate the performance of the file system shield with micro-benchmarks. We use IOZone [34] to subject the shield to random and sequential reads/writes. We compare the throughput of three different IOZone versions: (i) native glibc accessing a `tmpfs` file system; (ii) SCONE with the ephemeral file system without protection; and (iii) SCONE with an encrypted ephemeral file system.

Figure 12 shows the measured throughput as we vary the record size. We observe that IOZone on an ephemeral file system achieves a higher throughput than the native glibc IOZone on `tmpfs`. This is because the application does not issue any system calls when accessing data on the ephemeral file system—instead, it accesses the untrusted memory directly, without exiting the enclave. Enabling encryption on the ephemeral file system reduces the throughput by an order of magnitude.

In addition to the synthetic IOZone benchmark, we also measure how the shield impacts the performance of SQLite. We compare four versions: (i) SQLite with no protection; (ii) SQLCipher [54], which is SQLite with application level encryption; (iii) shielded SQLite on an encrypted ephemeral file system; and (iv) SQLCipher on an ephemeral file system (no authentication or encryption). The shielded versions use the memory-backed ephemeral file system, whereas the glibc versions of SQLite and SQLCipher use the standard Linux `tmpfs`. All protected versions use 256-bit keys.

Figure 13 shows the result of the SQLite benchmark. With small datasets (1000 keys), no cryptographic operations are necessary because the working set fits into SQLite's in-memory cache. This results in comparable performance across all versions. With bigger datasets, however, performance of the different versions diverge from the baseline, as SQLite starts to persist data on the file system resulting in an increasing number of cryptographic operations. We observe that the achieved throughput of the SQLCipher versions (approx. 60,000 operations per second) is about 35% of the baseline (approx. 170,000 operations per second), while the shielded SQLite version (approx. 140,000 operations per second) reaches about 80%. This is because the file

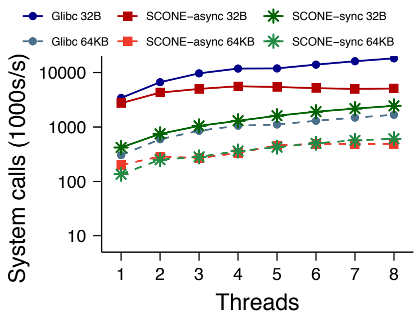


Figure 14: Frequency of system calls with asynchronous system call support

system shield of SCONE uses AES-GCM encryption, which outperforms AES in cipher block chaining (CBC) mode as used by default in SQLCipher. CBC mode restricts parallelism and requires an additional authentication mechanism.

4.4 Asynchronous system calls

Similar to Figure 2, Figure 14 shows how many `pwrite` calls can be executed by SCONE-async, SCONE-sync and natively. The x-axis refers to the number of OS-visible enclave threads for SCONE-async and SCONE-sync; for glibc, this is the number of native Linux threads. We vary the buffer size to see how the copy overhead influences the system call frequency. Larger buffers increase the overhead to move system call parameters from the enclave to the outside memory (§3.4); smaller buffers stress the shared memory queues to pass system call data.

For one OS thread, SCONE-async reaches almost the same number of system calls per second as glibc. Further scalability is likely to be possible by specialising our implementation of the lock-free FIFO queue.

5 Related Work

We discuss (i) software approaches that protect applications from privileged code, (ii) trusted hardware support and (iii) asynchronous system calls.

Software protection against privileged code. Protecting applications and their data from unauthorized access by privileged system software is a long-standing research objective. Initial work such as NGSCB [11, 43] and Proxos [57] executes untrusted and trusted OSs side-by-side using virtualization, with security-sensitive applications hosted by the trusted OS.

Subsequent work, including Overshadow [13], SP^3 [64], InkTag [27] and Virtual Ghost [16], has focused on reducing the size of the TCB by directly protecting application memory from unauthorized OS accesses. SEGO [36] extends these approaches by securing data handling inside and across devices using trusted metadata. Minibox [37] is a hypervisor-based sandbox that provides two-way protection between native applications and the guest OS. Unlike SCONE, all of these systems assume a trusted virtualization layer and struggle to protect applications from an attacker with physical access to the machine or who controls the virtualization layer.

Trusted hardware can protect security-sensitive applications, and implementations differ in their performance, commoditization, and security functionality.

Secure co-processors [39] offer tamper-proof physical isolation and can host arbitrary functionality. However, they are usually costly and limited in processing power. While in practice used to protect high-value secrets such as cryptographic keys [51], Bajaj and Sion [4, 5] demonstrate that secure co-processors can be used to split a database engine into trusted and untrusted parts. SCONE instead focuses on securing entire commodity container workloads and uses SGX to achieve better performance.

Trusted platform modules (TPM) [59] offer tailored services for securing commodity systems. They support remote attestation, size-restricted trusted storage and sealing of application data. Flicker [41] enables the multiplexing of secure modules that are integrity protected by the TPM. What limits Flicker's usability for arbitrary applications is the high cost of switching between secure and untrusted processing modes due to the performance limitations of current TPM implementations. TrustVisor [40] and CloudVisor [66] avoid the problem of frequent TPM usage by including the hypervisor within the TCB using remote attestation. This virtualization layer increases the size of the TCB, and neither solution can protect against an attacker with physical access to the machine's DRAM.

ARM TrustZone [3] has two system personalities, *secure* and *normal world*. This split meets the needs of mobile devices in which a rich OS must be separated from the system software controlling basic operations. Santos et al. [49] use TrustZone to establish trusted components for securing mobile applications. However, isolation of mutually distrustful components requires a trusted language runtime in the TCB because there is only a single secure world. TrustZone also does not protect against attackers with physical DRAM access.

As we described in §2.3, *Intel SGX* [29] offers fine-grained confidentiality and integrity at the enclave level. However, unlike TrustZone's secure world, enclaves cannot execute privileged code. Along the lines of the original SGX design goals of protecting tailored code for specific security-sensitive tasks [26], Intel provides an SDK [32, 33] to facilitate the implementation of simple enclaves. It features an interface definition language together with a code generator and a basic enclave library. Unlike SCONE, the SDK misses support for system calls and offers only restricted functionality inside the enclave.

Haven [6] aims to execute unmodified legacy Windows applications inside SGX enclaves by porting a Windows library OS to SGX. Relative to the limited EPC size of current SGX hardware, the memory requirements of a library OS are large. In addition, porting a complete library OS with a TCB containing millions of LOC also results in a large attack surface. By using only a modified C standard library, SCONE targets the demands of Linux containers, keeping the TCB small and addressing current SGX hardware constraints. Using asynchronous system calls, SCONE reduces enclave transition costs and puts emphasis on securing file and network communication for applications that do not use encryption.

VC3 [50] uses SGX to achieve confidentiality and integrity as part of the MapReduce programming model. VC3 jobs follow the executor interface of Hadoop but are not permitted to perform system calls. SCONE focuses on generic system support for container-based, interactive workloads but could be used as a basis for VC3 jobs that require extended system functionality.

Asynchronous system calls. FlexSC [52] batches system calls, reducing user/kernel transitions: when a batch is available, FlexSC signals the OS. In SCONE, application threads place system calls into a shared queue instead, which permits the OS threads to switch to other threads and stay inside the enclave. Moreover, SCONE uses a kernel module to execute system calls, while FlexSC requires invasive changes to the Linux kernel. Earlier work such as ULRPC [7] improves the performance of inter-process communication (IPC) using asynchronous, cross address space procedure calls via shared memory. In contrast, SCONE uses asynchronous system calls for all privileged operations, not just IPC.

6 Conclusion

SCONE increases the confidentiality and integrity of containerized services using Intel SGX. The secure containers of SCONE feature a TCB of only 0.6×–2× the application code size and are compatible with Docker. Using asynchronous system calls and a kernel module, SGX-imposed enclave transition overheads are reduced effectively. For all evaluated services, we achieve at least 60% of the native throughput; for Memcached, the throughput with SCONE is even higher than with native execution. At the same time, SCONE does not require changes to applications or the Linux kernel besides static recompilation of the application and the loading of a kernel module.

Acknowledgments. The authors thank the anonymous reviewers and our shepherd, Jeff Chase, for their valuable feedback. The work has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreements 645011 (SERECA) and 690111 (SecureCloud), and from the UK Engineering and Physical Sciences Research Council (EPSRC) under the CloudSafetyNet project (EP/K008129).

References

[1] Abramson, D., Jackson, J., Muthrasanallur, S., Neiger, G., Regnier, G., Sankaran, R., Schoinas, I., Uhlig, R., Vembu, B., and Wiegert, J. Intel Virtualization Technology for Directed I/O. *Intel Technology Journal* (2006).

[2] Anati, I., Gueron, S., Johnson, S., and Scarlata, V. Innovative Technology for CPU Based Attestation and Sealing. In *HASP* (2013).

[3] ARM Limited. *ARM Security Technology - Building a Secure System using TrustZone Technology*, 2009.

[4] Bajaj, S., and Sion, R. TrustedDB: A Trusted Hardware Based Database with Privacy and Data Confidentiality. In *SIGMOD* (2011).

[5] Bajaj, S., and Sion, R. CorrectDB: SQL Engine with Practical Query Authentication. *VLDB* (2013).

[6] Baumann, A., Peinado, M., and Hunt, G. Shielding Applications from an Untrusted Cloud with Haven. In *OSDI* (2014).

[7] Bershad, B. N., Anderson, T. E., Lazowska, E. D., and Levy, H. M. User-level interprocess communication for shared memory multiprocessors. *ACM TOCS 9*, 2 (May 1991), 175–198.

[8] Brewer, E. A. Kubernetes and the Path to Cloud Native. In *SoCC* (2015).

[9] Brickell, E., Graunke, G., Neve, M., and Seifert, J.-P. Software mitigations to hedge AES against cache-based software side channel vulnerabilities. *IACR Cryptology ePrint Archive 2006* (2006), 52.

[10] Brown, N. Linux Kernel Overlay Filesystem Documentation. `https://www.kernel.org/doc/Documentation/filesystems/overlayfs.txt`, 2015.

[11] Carroll, A., Juarez, M., Polk, J., and Leininger, T. Microsoft Palladium: A Business Overview. *Microsoft Content Security Business Unit* (2002), 1–9.

[12] Checkoway, S., and Shacham, H. Iago Attacks: Why the System Call API is a Bad Untrusted RPC Interface. In *ASPLOS* (2013).

[13] Chen, X., Garfinkel, T., Lewis, E. C., Subrahmanyam, P., Waldspurger, C. A., Boneh, D., Dwoskin, J., and Ports, D. R. Overshadow: A Virtualization-based Approach to Retrofitting Protection in Commodity Operating Systems. In *ASPLOS* (2008).

[14] Cooper, B. F., Silberstein, A., Tam, E., Ramakrishnan, R., and Sears, R. Benchmarking Cloud Serving Systems with YCSB. In *SoCC* (2010).

[15] Costan, V., and Devadas, S. Intel SGX explained. Tech. rep., Cryptology ePrint Archive, Report 2016/086, 2016.

[16] Criswell, J., Dautenhahn, N., and Adve, V. Virtual Ghost: Protecting Applications from Hostile Operating Systems. In *ASPLOS* (2014).

[17] CVE-ID: CVE-2014-9357. Available from MITRE at `https://cve.mitre.org`, Dec. 2014.

[18] CVE-ID: CVE-2015-3456. Available from MITRE at `https://cve.mitre.org`, May 2015.

[19] CVE-ID: CVE-2015-5154. Available from MITRE at `https://cve.mitre.org`, Aug. 2015.

[20] DIERKS, T., AND RESCORLA, E. The Transport Layer Security (TLS) Protocol Version 1.2. RFC 5246 (Proposed Standard), Aug. 2008. Updated by RFCs 5746, 5878, 6176, 7465, 7507, 7568, 7627, 7685.

[21] DONG, Y., YANG, X., LI, J., LIAO, G., TIAN, K., AND GUAN, H. High performance network virtualization with SR-IOV. *Journal of Parallel and Distributed Computing 72*, 11 (2012), 1471–1480.

[22] FELTER, W., FERREIRA, A., RAJAMONY, R., AND RUBIO, J. An updated performance comparison of virtual machines and Linux containers. In *ISPASS* (2015).

[23] FITZPATRICK, B. Distributed caching with memcached. *Linux Journal* (Aug. 2004).

[24] GRABER, H. LXC Linux Containers, 2014.

[25] HAPROXY. `http://www.haproxy.org`, 2016.

[26] HOEKSTRA, M., LAL, R., PAPPACHAN, P., PHEGADE, V., AND DEL CUVILLO, J. Using Innovative Instructions to Create Trustworthy Software Solutions. In *HASP* (2013).

[27] HOFMANN, O. S., KIM, S., DUNN, A. M., LEE, M. Z., AND WITCHEL, E. InkTag: Secure Applications on an Untrusted Operating System. In *ASPLOS* (2013).

[28] INITIATIVE, T. O. C. `https://www.opencontainers.org`, 2016.

[29] INTEL CORP. Software Guard Extensions Programming Reference, Ref. 329298-002US. `https://software.intel.com/sites/default/files/managed/48/88/329298-002.pdf`, Oct. 2014.

[30] INTEL CORP. Intel Software Guard Extensions (Intel SGX), Ref. 332680-002. `https://software.intel.com/sites/default/files/332680-002.pdf`, June 2015.

[31] INTEL CORP. Product Change Notification 114074-00. `https://qdms.intel.com/dm/i.aspx/5A160770-FC47-47A0-BF8A-062540456F0A/PCN114074-00.pdf`, October 2015.

[32] INTEL CORP. Intel Software Guard Extensions for Linux OS. `https://01.org/intel-softwareguard-extensions`, June 2016.

[33] INTEL CORP. Intel Software Guard Extensions (Intel SGX) SDK. `https://software.intel.com/sgx-sdk`, 2016.

[34] IOZONE. `http://www.iozone.org`, 2016.

[35] KUBERNETES. `http://kubernetes.io`, 2016.

[36] KWON, Y., DUNN, A. M., LEE, M. Z., HOFMANN, O. S., XU, Y., AND WITCHEL, E. Sego: Pervasive Trusted Metadata for Efficiently Verified Untrusted System Services. In *ASPLOS* (2016).

[37] LI, Y., MCCUNE, J., NEWSOME, J., PERRIG, A., BAKER, B., AND DREWRY, W. MiniBox: A Two-Way Sandbox for x86 Native Code. In *ATC* (2014).

[38] LIBC, M. `https://www.musl-libc.org`, 2016.

[39] LINDEMANN, M., PEREZ, R., SAILER, R., VAN DOORN, L., AND SMITH, S. Building the IBM 4758 Secure Coprocessor. *Computer 34*, 10 (Oct 2001), 57–66.

[40] MCCUNE, J. M., LI, Y., QU, N., ZHOU, Z., DATTA, A., GLIGOR, V., AND PERRIG, A. TrustVisor: Efficient TCB Reduction and Attestation. In *S&P* (2010).

[41] MCCUNE, J. M., PARNO, B. J., PERRIG, A., REITER, M. K., AND ISOZAKI, H. Flicker: An Execution Infrastructure for TCB Minimization. In *EuroSys* (2008).

[42] MERKEL, D. Docker: Lightweight Linux Containers for Consistent Development and Deployment. *Linux Journal* (Mar. 2014).

[43] PEINADO, M., CHEN, Y., ENGLAND, P., AND MANFERDELLI, J. NGSCB: A Trusted Open System. In *ACISP* (2004).

[44] PROJECT, A. H. S. `https://httpd.apache.org`, 2016.

[45] PURDILA, O., GRIJINCU, L. A., AND TAPUS, N. LKL: The Linux kernel library. In *RoEduNet* (2010).

[46] REDIS. `http://redis.io`, 2016.

[47] REESE, W. Nginx: the High-Performance Web Server and Reverse Proxy. *Linux Journal* (Sept. 2008).

[48] RKT (COREOS). `https://coreos.com/rkt`, 2016.

[49] SANTOS, N., RAJ, H., SAROIU, S., AND WOLMAN, A. Using ARM TrustZone to build a trusted language runtime for mobile applications. In *ASPLOS* (2014).

[50] SCHUSTER, F., COSTA, M., FOURNET, C., GKANTSIDIS, C., PEINADO, M., MAINAR-RUIZ, G., AND RUSSINOVICH, M. VC3: Trustworthy Data Analytics in the Cloud using SGX. In *S&P* (2015).

[51] SERVICES, A. W. AWS CloudHSM Getting Started Guide. `http://aws.amazon.com/cloudhsm`, 2016.

[52] SOARES, L., AND STUMM, M. FlexSC: Flexible System Call Scheduling with Exception-less System Calls. In *OSDI* (2010).

[53] SOLTESZ, S., PÖTZL, H., FIUCZYNSKI, M. E., BAVIER, A., AND PETERSON, L. Container-based operating system virtualization: A scalable, high-performance alternative to hypervisors. *SIGOPS OSR* (Mar. 2007).

[54] SQLCIPHER. `https://www.zetetic.net/sqlcipher`, 2016.

[55] SQLite. `https://www.sqlite.org`, 2016.

[56] SWARM, D. `https://docs.docker.com/swarm`, 2016.

[57] TA-MIN, R., LITTY, L., AND LIE, D. Splitting Interfaces: Making Trust Between Applications and Operating Systems Configurable. In *OSDI* (2006).

[58] THONES, J. Microservices. *IEEE Software 32*, 1 (2015), 116–116.

[59] TRUSTED COMPUTING GROUP. Trusted Platform Module Main Specification, version 1.2, revision 116, 2011.

[60] UHLIG, R., NEIGER, G., RODGERS, D., SANTONI, A. L., MARTINS, F., ANDERSON, A. V., BENNETT, S. M., KÄGI, A., LEUNG, F. H., AND SMITH, L. Intel virtualization technology. *Computer 38*, 5 (2005), 48–56.

[61] WONG, W. Stunnel: SSLing Internet Services Easily. *SANS Institute, November* (2001).

[62] A HTTP benchmarking tool based mostly on wrk. `https://github.com/giltene/wrk2`, 2016.

[63] XU, Y., CUI, W., AND PEINADO, M. Controlled-Channel Attacks: Deterministic Side Channels for Untrusted Operating Systems. In *S&P* (2015).

[64] YANG, J., AND SHIN, K. G. Using Hypervisor to Provide Data Secrecy for User Applications on a Per-page Basis. In *VEE* (2008).

[65] ZETTER, K. NSA Hacker Chief Explains How to Keep Him Out of Your System. *Wired* (Jan. 2016).

[66] ZHANG, F., CHEN, J., CHEN, H., AND ZANG, B. CloudVisor: Retrofitting Protection of Virtual Machines in Multi-tenant Cloud with Nested Virtualization. In *SOSP* (2011).

Coordinated and Efficient Huge Page Management with Ingens

Youngjin Kwon, Hangchen Yu, Simon Peter, Christopher J. Rossbach[1], Emmett Witchel
The University of Texas at Austin
[1]The University of Texas at Austin and VMware Research Group

Abstract

Modern computing is hungry for RAM, with today's enormous capacities eagerly consumed by diverse workloads. Hardware address translation overheads have grown with memory capacity, motivating hardware manufacturers to provide TLBs with thousands of entries for large page sizes (called huge pages). Operating systems and hypervisors support huge pages with a hodge-podge of best-effort algorithms and spot fixes that made sense for architectures with limited huge page support, but the time has come for a more fundamental redesign.

Ingens is a framework for huge page support that relies on a handful of basic primitives to provide transparent huge page support in a principled, coordinated way. By managing contiguity as a first-class resource and by tracking utilization and access frequency of memory pages, Ingens is able to eliminate a number of fairness and performance pathologies that plague current systems. Experiments with our prototype demonstrate fairness improvements, performance improvements (up to 18%), tail-latency reduction (up to 41%), and reduction of memory bloat from 69% to less than 1% for important applications like Web services (e.g., the Cloudstone benchmark) and the Redis key-value store.

1 Introduction

Modern computing platforms can support terabytes of RAM and workloads able to take advantage of such large memories are now commonplace [51]. However, increased capacity represents a significant challenge for address translation. All modern processors use page tables for address translation and TLBs to cache virtual-to-physical mappings. Because TLB capacities cannot scale at the same rate as DRAM, TLB misses and address translation can incur crippling performance penalties for large memory workloads [44, 53] when these workloads use traditional page sizes (i.e., 4KB). Hardware-supported address virtualization (e.g., AMD's nested page tables) increases average-case address translation overhead because multi-dimensional page tables amplify worst-case translation costs by 6× [59]. Hardware manufacturers have addressed increasing DRAM capacity with better support for larger page sizes, or *huge pages*, which reduce address translation overheads by reducing the frequency of TLB misses. However, the success of these mechanisms is critically dependent on the ability of the operating systems and hypervisors to manage huge pages.

While huge pages have been commonly supported in hardware since the 90s [75, 76], until recently, processors have had a very small number of TLB entries reserved for huge pages, limiting their usability. Newer architectures support thousands of huge page entries in dual-level TLBs (e.g., 1,536 in Intel's Skylake [1]), which is a major change: the onus of better huge page support has shifted from the hardware to the system software. There is now both an urgent need and an opportunity to modernize memory management.

Operating system memory management has generally responded to huge page hardware with best-effort algorithms and spot fixes, choosing to keep their management algorithms focused on the 4KB page (which we call a *base page*). For example, Linux and KVM (Linux's in-kernel hypervisor) adequately support many large-memory workloads (i.e., ones with simple, static memory allocation behavior), but a variety of common workloads are exposed to unacceptable performance overheads, wasted memory capacity, and unfair performance variability when using huge pages. These problems are common and severe enough that administrators generally disable huge pages (e.g., MongoDB, Couchbase, Redis, SAP, Splunk, etc.) despite their obvious average-case performance advantages [24, 9, 11, 30, 26, 32, 34, 37]. Other operating systems have similar or even more severe problems supporting huge pages (see §2.2 and §3.4).

Ingens[1] is a memory manager for the operating system and hypervisor that replaces the best-effort mechanisms

[1]Ingens is Latin for huge.

and spot-fixes of the past with a coordinated, unified approach to huge pages; one that is better targeted to the increased TLB capacity in modern processors. Ingens does not interfere with workloads that perform well with current huge page support: the prototype adds 0.7% overhead on average (Table 4). Ingens addresses the following problems endemic to current huge page support, and we quantify the impact of these problems on real workloads using our prototype.

- **Latency.** Huge pages expose applications to high latency variation and increased tail latency (§3.1). Ingens improves the Cloudstone benchmark [77] by 18% and reduces 90th percentile tail-latency by 41%.
- **Bloat.** Huge pages can make a process or virtual machine (VM) occupy a large amount of physical memory while much of that memory remains unusable due to internal fragmentation (§3.2). For Redis, Linux bloats memory use by 69%, while Ingens bloats by just 0.8%.
- **Unfairness.** Simple, greedy allocation of huge pages is unfair, causing large and persistent performance variation across identical processes or VMs (§3.5). Ingens makes huge page allocation fair (e.g., Figure 5).
- **High-performance memory savings.** Services that reduce memory consumption, such as kernel same-page merging (KSM), can prevent a VM from using huge pages (§3.6). On one workload (Figure 11), Linux saves 9.2% of memory but slows down the programs by 6.8–19%. Ingens saves 71.3% of the memory that Linux/KVM can save with only a 1.5–2.6% slowdown.

Ingens is a memory management redesign that brings performance, memory savings and fairness to memory-intensive applications with dynamic memory behavior. It is based on two principles: (1) memory contiguity is an explicit resource to be allocated across processes and (2) good information about spatial and temporal access patterns is essential to managing contiguity; it allows the OS to tell/predict when contiguity is/will be profitably used. The measured performance of the Ingens prototype on realistic workloads validates the approach.

2 Background

Current trends in memory management hardware are making it critical that system software support huge pages efficiently and flexibly. This section considers those trends along with the challenges huge page support creates for the OS and hypervisor. We provide an overview of huge page support in modern operating systems and conclude with experiments that show the performance benefits for the state-of-the-art in huge page management.

2.1 Virtual memory hardware trends

Virtual memory decouples the address space used by programs from that exported by physical memory (RAM). A page table maps virtual to physical page number, with recently used page table entries cached in the hardware translation lookaside buffer (TLB). Increasing the page size increases TLB *reach* (the amount of data covered by translations cached in the TLB), but larger pages require larger regions of contiguous physical memory. Large pages can suffer from internal fragmentation (unused portions within the unit of allocation) and can also increase external fragmentation (reducing the remaining supply of contiguous physical memory). Using larger pages requires more active memory management from the system software to increase available contiguity and avoid fragmentation.

Seminal work in huge page management recognized the importance of explicitly managing memory contiguity in the OS [68] and formed the basis for huge page support in FreeBSD. Innovations of Ingens relative to previous work are considered in detail in Section 3.4; here we survey recent hardware trends that make the need for system support of huge pages more urgent.

DRAM Growth. Larger DRAM sizes have led to deeper page tables, increasing the number of memory references needed to look up a virtual page number. x86 uses a 4-level page table with a worst case of four page table memory references to perform a single address translation.

Hardware memory virtualization. Extended page tables (Intel) or nested page tables (AMD) require additional indirection for each stage of memory address translation, making the process of resolving a virtual page number even more complex. With extended page tables, both the guest OS and host hypervisor perform virtual to physical translations to satisfy a single request. During translation, guest physical addresses are treated as host virtual addresses, which use hardware page-table walkers to perform the entire translation. Each layer of lookup in the guest can require a multi-level translation in the host, amplifying the maximum cost to 24 lookups [59, 40], and increasing average latencies [67].

Increased TLB reach. Recently, Intel has moved to a two-level TLB design, and in the past few years has provided a significant number of second-level TLB entries for huge pages, going from zero for Sandy Bridge and Ivy Bridge to 1,024 for Haswell [2] (2013) and 1,536 for Skylake [1] (2015).

Better hardware support for multiple page sizes creates an opportunity for the OS and the hypervisor, but it puts stress on the current memory management algorithms. In addition to managing the complexity of different page granularities, system software must generate and maintain significant memory contiguity to use larger page sizes.

Name	Suite/Application	Description
429.mcf	SPEC CPU 2006 [33]	Single-threaded scientific computation
Canneal	PARSEC 3.0 [28]	Parallel scientific computation
SVM [64]	Liblinear [22]	Machine learning, Support vector machine
Tunkrank [8]	PowerGraph [55]	Large scale in-memory graph analytics
Nutch [19]	Hadoop [4]	Web search indexing using MapReduce
MovieRecmd [25]	Spark/MLlib [5]	Machine learning, Movie recommendation
Olio	Cloudstone [8]	Social-event Web service (ngnix/php/mysql)
Redis	Redis [29]	In-memory Key-value store
MongoDB	MongoDB [23]	In-memory NoSQL database

Table 1: Summary of memory intensive workloads.

Issue	OS	Hyp
Page fault latency (§3.1)	O	
Bloat (§3.2)	O	
Fragmentation (§3.3)	O	O
Unfair allocation (§3.5)	O	O
Memory sharing (§3.6)		O

Table 2: Summary of issues in Linux as the guest OS and KVM as the host hypervisor.

2.2 Operating system support for huge pages

Early operating system support for huge pages provided a separate interface for explicit huge page allocation from a dedicated huge page pool configured by the system administrator. Windows and OS X continue to have this level of support. In Windows, applications must use an explicit memory allocation API for huge page allocation [21] and Windows recommends that applications allocate huge pages all at once when they begin. OS X applications also must set an explicit flag in the memory allocation API to use huge pages [15].

Initial huge page support in Linux used a similar separate interface for huge page allocation that a developer must invoke explicitly (called `hugetlbfs`). Developers did not like the burden of this alternate API and kernel developers wanted to bring the benefits of huge pages to legacy applications and applications with dynamic memory behavior [6, 36]. Hence, the primary way huge pages are allocated in Linux today is *transparently* by the kernel.

Transparent support is vital. Transparent huge page support [80, 68] is the only practical way to bring the benefits of huge pages to all applications, which can remain unchanged while the system provides them with the often significant performance advantages of huge pages. With transparent huge page support, the kernel allocates memory to applications using base pages. We say the kernel **promotes** a sequence of 512 properly aligned pages to a huge page (and demotes a huge page into 512 base pages).

Transparent management of huge pages best supports the multi-programmed and dynamic workloads typical of web applications and analytics where memory is contended and access patterns are often unpredictable. To the contrary, when a single big-memory application is the only important program running, the application can simply map a large region and keep it mapped for the duration of execution, for example fast network functions using Intel's Data Plane Development Kit [10]. These simple programs are well supported by even the rudimentary huge page support in Windows and OS X. However, multi-programmed workloads and workloads with more complex memory behavior are common in enterprise and cloud computing, so Ingens focuses on OS support for these more challenging cases. While transparent huge page support is far more developer-friendly than explicit allocation, it creates memory management challenges in the operating system that Ingens addresses.

Linux running on Intel processors currently has the best transparent huge page support among commodity OSes so we base our prototype on it and most of our discussion focuses on Linux. We quantify Linux's performance advantages in Section 3.4. The design of Ingens focuses on 4 KB (base) and 2 MB (huge) pages because these are most useful to applications with dynamic memory behavior (1 GB are usually too large for user data structures).

Linux is greedy and aggressive. Linux's huge page management algorithms are greedy: it promotes huge pages in the page fault handler based on local information. Linux is also aggressive: it will always try to allocate a huge page. Huge pages require 2 MB of contiguous free physical memory but sometimes contiguous physical memory is in short supply (e.g., when memory is fragmented). Linux's approach to huge page allocation works well for simple applications that allocate a large memory region and use it uniformly, but we demonstrate many applications that have more complex behavior and are penalized by Linux's greedy and aggressive promotion of huge pages (§3). Ingens recognizes that memory contiguity is a valuable resource and explicitly manages it.

2.3 Hypervisor support for huge pages

Ingens focuses on the case where Linux is used both as the guest operating system and as the host hypervisor (i.e., KVM [62]). The Linux/KVM pair is widely used in cloud deployments [27, 16, 3]. In the hypervisor, Ingens supports host huge pages mapped from guest physical memory. When promoting guest physical memory, Ingens modifies the extended page table to use huge pages because it is acting as a hypervisor, not as an operating

Workloads	h_B g_H	h_H g_B	h_H g_H
429.mcf	1.18	1.13	1.43
Canneal	1.11	1.10	1.32
SVM	1.14	1.17	1.53
Tunkrank	1.11	1.11	1.30
Nutch	1.01	1.07	1.12
MovieRecmd	1.03	1.02	1.11
Olio	1.43	1.08	1.46
Redis	1.12	1.04	1.20
MongoDB	1.08	1.22	1.37

Table 3: Application speed up for huge page (2 MB) support relative to host (h) and guest (g) using base (4 KB) pages. For example, h_B means the host uses base pages and h_H means the host uses both base and huge pages.

system.

Because operating system and hypervisor memory management are unified in Linux, Ingens adopts the unified model. Some of the problems with huge pages that we describe in Section 3 only apply to the OS and some only to the hypervisor (summarized in Table 2). For example, addressing memory sharing vs. performance (§3.6) requires only hypervisor modifications and would be as successful for a Windows guest as it is for a Linux guest. We leave for future work determining the most efficient way to implement Ingens for operating systems and hypervisors that do not share memory management code.

2.4 Performance improvement from huge pages

Table 1 describes a variety of memory-intensive real-world applications including web infrastructure such as key/value stores and databases, as well as scientific applications, data analytics and recommendation systems. Measurements with hardware performance counters show they all spend a significant portion of their execution time doing page walks. For example, when using base pages for both guest and host, we measure 429.mcf spending 47.5% of its execution time doing page walks (24.2% for the extended page table and 23.3% for the guest page table). On the other hand, 429.mcf spends only 4.2% of its execution time walking page tables when using huge pages for both the guest and host.

We execute all workloads in a KVM virtual machine running Linux with default transparent huge page support [80] for both the application (in the guest OS) and the virtual machine (in the host OS). The hardware configuration is detailed in Section 6.

Table 3 shows the performance improvements gained with transparent huge page support for both the guest and the host operating system. The table shows speedup normalized to the case where both host and guest use only base pages. In every case, huge page support helps performance, often significantly (up to 53%). The largest speedup is always attained when both host and guest use huge pages.

These results show the value of huge page support and show that Linux's memory manager can obtain that benefit under simple operating conditions. However, a variety of more challenging circumstances expose the limitations of Linux's memory management.

3 Current huge page problems

This section quantifies the limitations in performance and fairness for the state-of-the-art in transparent huge page management. We examine virtualized systems with Linux/KVM as the guest OS and hypervisor. The variety and severity of the limitations motivate our redesign of page management. All data is collected using the experimental setup described in Section 2.4.

3.1 Page fault latency and synchronous promotion

When a process faults on an anonymous memory region, the page fault handler allocates physical memory to back the page. Both base and huge pages share this code path. Linux is greedy and aggressive in its allocation of huge pages, so if an application faults on a base page, Linux will immediately try to upgrade the request and allocate a huge page if it can.

This greedy approach fundamentally increases page fault latency for two reasons. First, Linux must zero pages before returning them to the user. Huge pages are 512$\times$ larger than base pages, and thus are much slower to clear. Second, huge page allocation requires 2 MB of physically contiguous memory. When memory is fragmented, the OS often must compact memory to generate that much contiguity. Previous work shows that memory quickly fragments in multi-tenant cloud environments [41]. When memory is fragmented, Linux will often synchronously compact memory in the page fault handler, increasing average and tail latency.

To measure these effects, we compare page fault latency when huge pages are enabled and disabled, in fragmented and non-fragmented settings. We quantify fragmentation using the *free memory fragmentation index* (FMFI) [58], a value between 0 (unfragmented) and 1 (highly fragmented). A microbenchmark maps 10 GB of anonymous virtual memory and reads it sequentially.

When memory is unfragmented ($\text{FMFI} < 0.1$), page clearing overheads increase average page fault latency from 3.6 μs for base pages only to 378 μs for huge pages (105$\times$ slower). When memory is heavily fragmented, ($\text{FMFI} = 0.9$), the 3.6 μs average latency for base pages grows to 8.1 μs (2.1$\times$ slower) for base and huge pages. Average latency is lower in the fragmented case because 98% of the allocations fall back to base pages (e.g. because memory is too fragmented to allocate a huge page).

SVM	Synchronous	Asynchronous
Exec. time (sec)	178 (1.30×)	228 (1.02×)
Huge page	4.8 GB	468 MB
Promotion speed	immediate	1.6 MB/s

Table 4: Comparison of synchronous promotion and asynchronous promotion when both host and guest use huge pages. The parenthesis is speedup compared to not using huge pages. We use the default asynchronous promotion speed of Ubuntu 14.04.

Workload	Using huge pages	Not using huge pages
Redis	20.7 GB (1.69×)	12.2 GB
MongoDB	12.4 GB (1.23×)	10.1 GB

Table 5: Physical memory size of Redis and MongoDB.

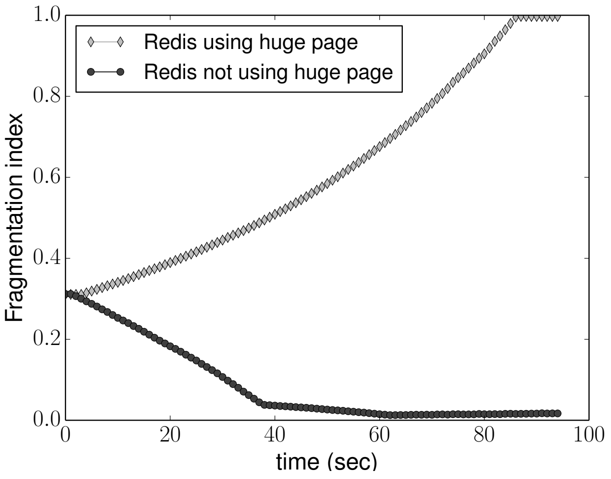


Figure 1: Fragmentation index in Linux when running a Redis server, with Linux using (and not using) huge pages. The System has 24 GB memory. Redis uses 13 GB, other processes use 5 GB, and system has 6 GB free memory.

Compacting and zeroing memory in the page fault handler penalizes applications that are sensitive to average latency and to tail latency, such as Web services.

To avoid this additional page fault latency, Linux can promote huge pages asynchronously, based on a configurable asynchronous promotion speed (in MB/s). Table 4 shows performance measurements for asynchronous-only huge page promotion when executing SVM in a virtual machine. Asynchronous-only promotion turns a 30% speedup into a 2% speedup: it does not promote fast enough. Simply increasing the promotion speed does not solve the problem. Earlier implementations of Linux did more aggressive asynchronous promotion, incurring unacceptably high CPU utilization for memory scanning and compaction. The CPU use of aggressive promotion reduced or in some cases erased the performance benefits of huge pages, causing users to disable transparent huge page support in practice [17, 14, 13, 7].

3.2 Increased memory footprint (bloat)

Huge pages improve performance, but applications do not always fully utilize the huge pages allocated to them. Linux greedily allocates huge pages even though underutilized huge pages create internal fragmentation. A huge page might eliminate TLB misses, but the cost is that a process using less than a full huge page has to reserve the entire region.

Table 5 shows memory bloat from huge pages when running Redis and MongoDB, each within their own virtual machine. For Redis, we populate 2 million keys with 8 KB objects and then delete 70% of the keys randomly. Redis frees the memory backing the deleted objects which leaves physical memory sparsely allocated. Linux promotes the sparsely allocated memory to huge pages, creating internal fragmentation and causing Redis to use 69% more memory compared to not using huge pages. We demonstrate the same problem in MongoDB, making 10 million `get` requests for 15 million 1 KB objects which are initially in persistent storage. MongoDB allocates the objects sparsely in a large virtual address space. Linux promotes huge pages including unused memory, and as a result, MongoDB uses 23% more memory relative to running without huge page support.

Greedy and aggressive allocation of huge pages makes it impossible to predict an application's total memory usage in production because memory usage depends on huge page use, which in turn depends on memory fragmentation and the allocation pattern of applications. Table 5 shows if an administrator provisions 18 GB memory (1.5× over-provisioning relative to using only base pages), Redis starts swapping when it uses huge pages, negating the benefits of caching objects in memory [31].

While these experiments illustrate the potential impact of bloat for a handful of workloads, it is important to note that the problem is fundamental to Linux's current design. Memory bloating can happen in any working set, memory, and TLB size: application-level memory usage can conspire with aggressive promotion to create internal fragmentation that the OS cannot address. In such situations, such applications will eventually put the system under memory pressure regardless of physical memory size.

3.3 Huge pages increase fragmentation

One common theme in analyzing page fault latency (§3.1) and memory bloat (§3.2) is Linux's greedy allocation and promotion of huge pages. We now measure how aggressive promotion of huge pages quickly consumes available physical memory contiguity, which then increases memory fragmentation for the remaining physical memory.

OS	SVM	Canneal	Redis
FreeBSD	1.28	1.13	1.02
Linux	1.30	1.21	1.15

Table 6: Performance speedup when using huge page in different operating systems.

Increasing fragmentation is the precondition for problems with page fault latency and memory bloat, so greedy promotion creates a vicious cycle. We again rely on the free memory fragmentation index, or FMFI to quantify the relationship between huge page allocation and fragmentation.

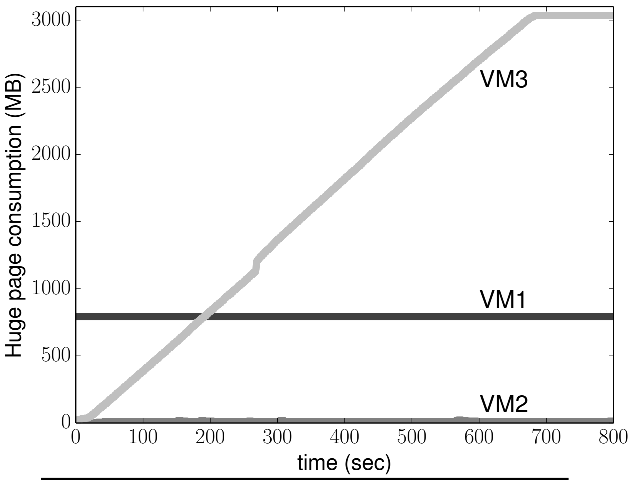
Figure 1 shows the fragmentation index over time when running the popular key-value store application Redis in a virtual machine. Initially, the system is lightly fragmented (FMFI = 0.3) by other processes. Through the measurement period, Redis clients populate the server with 13 GB of key/value pairs. Redis rapidly consumes contiguous memory as Linux allocates huge pages to it, increasing the fragmentation index. When the FMFI is equal to 1, the remaining physical memory is so fragmented, Linux starts memory compaction to allocate huge pages.

3.4 Comparison with FreeBSD huge page support

FreeBSD supports transparent huge pages using reservation-based huge page allocation [68]. When applications start accessing a 2 MB virtual address region, the page fault handler reserves contiguous memory, but does not promote the region to a huge page. It allocates base pages from the reserved memory for subsequent page faults in the region. FreeBSD monitors page utilization of the region and promotes it to a huge page only when all base pages of the reserved memory are allocated. FreeBSD is therefore slower to promote huge pages than Linux and promotion requires complete utilization of a 2 MB region.

FreeBSD supports huge pages for file-cached pages. x86 hardware maintains access/dirty bits for entire huge pages—any read or write will set the huge page's access/dirty bit. FreeBSD wants to avoid increasing IO traffic when evicting from the page cache or swapping. Therefore it is conservative about creating writable huge pages. When FreeBSD promotes a huge page, it marks it read-only, with writes demoting the huge page. Only when all pages in the region are modified will FreeBSD then promote the region to a writable huge page. The read-only promotion design does not increase IO traffic from the page cache because huge pages consist of either all clean (read-only) or all modified base pages.

FreeBSD promotion of huge pages is more conservative than in Linux, which reduces memory bloating, but yields slower performance. Table 6 compares the performance benefits of huge pages in FreeBSD and Linux. Applications with dense, uniform access memory patterns (e.g., SVM) enjoy similar speedups on Linux and FreeBSD. However, FreeBSD does not support asynchronous promotion, so applications which allocate memory gradually (e.g., Canneal) show less benefit. Redis makes frequent hash table updates and exhibits many read-only huge page demotions in FreeBSD. Consequently, Redis also shows limited speedup compared with Linux.



SVM	VM1	VM2	VM3
Exec. time (sec)	533 (1.12×)	589 (1.24×)	475

Figure 2: Unfair allocation of huge pages in KVM. Three virtual machines run concurrently, each executing SVM. The line graph is huge page size (MB) over time and the table shows execution time of SVM for 2 iterations.

3.5 Unfair performance

All of our measurements are on virtual machines where Linux is the guest operating system, and KVM (Linux's in-kernel hypervisor) is the host hypervisor. Ingens modifies the memory management code of both Linux and KVM. The previous sections focused on problems with operating system memory management, the remaining sections describe problems with KVM memory management.

Unfair huge page allocation can lead to unfair performance differences when huge pages become scarce. Linux does not fairly redistribute contiguity, which can lead to unfair performance imbalance. To demonstrate this problem, we run 4 virtual machines in a setting where memory is initially fragmented (FMFI = 0.85). Each VM uses 8 GB of memory. VM0 starts first and obtains all huge pages that are available (3 GB). Later, VM1 starts and begins allocating memory, during which VM2 and VM3 start. VM0 then terminates, releasing its 3 GB of huge pages. We measure how Linux redistributes that contiguity to the remaining identical VMs.

The graph in Figure 2 shows the amount of huge page

Policy	Mem saving	Performance slowdown	H/M
No sharing	–	429.mcf: 278 SVM: 191 Tunkrank: 236	429.mcf: 99% SVM: 99% Tunkrank: 99%
KVM (Linux)	1.19 GB (9.2%)	429.mcf: 331 (19.0%) SVM: 204 (6.8%) Tunkrank: 268 (13.5%)	429.mcf: 66% SVM: 90% Tunkrank: 69%
Huge page sharing	199 MB (1.5%)	429.mcf: 278 (0.0%) SVM: 194 (1.5%) Tunkrank: 238 (0.8%)	429.mcf: 99% SVM: 99% Tunkrank: 99%

Table 7: Memory saving and performance trade off for a multi-process workload. Each row is an experiment where all workloads run concurrently in separate virtual machines. H/M - huge page ratio out of total memory used. Parentheses in the Mem saving column expresses the memory saved as a percentage of the total memory (13 GB) allocated to all three virtual machines.

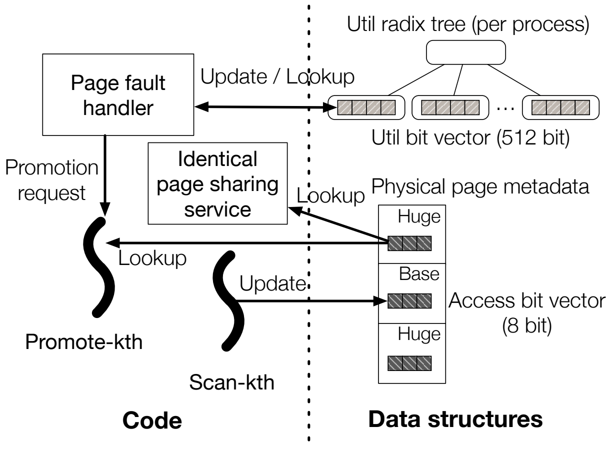


Figure 3: Important code and data structures in the Ingens memory manager.

memory allocated to VM1, VM2, and VM3 (all running SVM) over time, starting 10 seconds before the termination of VM0. When VM1 allocates memory, Linux compacts memory for huge page allocation, but compaction begins to fail at 810 MB. VM2 and VM3 start without huge pages. When VM0 terminates 10 seconds into the experiment, Linux allocates all 3 GB of recently freed huge pages to VM3 through asynchronous promotion. This creates significant and persistent performance inequality among the VMs. The table in Figure 2 shows the variation in performance (NB: to avoid IO measurement noise, data loading time is excluded from the measurement). In a cloud provider scenario, with purchased VM instances of the same type, users have good reason to expect similar performance from identical virtual machine instances, but VM2 is 24% slower than VM3.

3.6 Memory sharing vs. performance

Modern hypervisors detect and share memory pages from different virtual machines whose contents are identical [81, 63]. The ability to share identical memory reduces the memory consumed by guest VMs, increasing VM consolidation ratios. In KVM, identical page sharing in the host is done transparently in units of base pages. If the contents of a base page are duplicated in a different VM, but the duplicated base page is contained within a huge page, KVM will split the huge page into base pages to enable sharing. This policy prioritizes reducing memory footprint over preservation of huge pages, so it penalizes performance.

Another possible policy, which we call *huge page sharing*, would not split huge pages. A base page is not allowed to share pages belonging to a huge page to prevent the demotion of the huge page but it can share base pages. In contrast, a huge page is only allowed to share huge pages. We implement huge page sharing to compare with KVM and the result is shown in Table 7. We fit the virtual machine memory size to the working set size of each workload to avoid spurious sharing of zeroed pages. KVM saves 9.2% of memory but the workloads show a slowdown of up to 19.0% because TLB misses are increased by splitting huge pages (the percentage of huge pages in use (H/M) goes down to 66%). On the other hand, while huge page sharing preserves good performance, it provides only reduced memory consumption by 1.5%. This tradeoff between performance and memory savings is avoidable. Identical page sharing services can and should be coordinated with huge page management to obtain both performance and memory saving benefits.

4 Design

Ingens's goal is to enable transparent huge page support that reduces latency, latency variability and bloat while providing meaningful fairness guarantees and reasonable tradeoffs between high performance and memory savings. Ingens builds on a handful of basic primitives to achieve these goals: utilization tracking, access frequency tracking, and contiguity monitoring.

While the discussion in this section is mostly expressed in terms of process behavior, Ingens techniques apply equally to processes and to virtual machines. Figure 3 shows the major data structures and code paths of Ingens, which we describe in this section.

4.1 Monitoring space and time

Ingens unifies and coordinates huge page management by introducing two efficient mechanisms to measure the utilization of huge-page sized regions (space) and how frequently huge-page sized regions are accessed (time). Ingens collects this information efficiently and then leverages it throughout the kernel to make policy decisions, using two bitvectors. We describe both.

Util bitvector. The util bitvector records which base pages are used within each huge-page sized memory region (an aligned 2 MB region containing 512 base pages). Each bit set in the util bitvector indicates that the corresponding base page is in use. The bitvector is stored in a radix tree and Ingens uses a huge-page number as the key to lookup a bitvector. The page fault handler updates the util bitvector.

Access bitvector. The access bitvector records the recent access history of a process to its pages (base or huge). Scan-kth periodically scans a process' hardware access bits in its page table to maintain per-page (base or huge) access frequency information, stored as an 8-bit vector within Linux' page metadata. Ingens computes the exponential moving average (EMA) [12] from the bitvector which we define as follows:

$$F_t = \alpha(\mathit{weight(util\ bitvector)}) + (1-\alpha)F_{t-1} \quad (1)$$

The *weight* is the sum of set bits in the bitvector, F_t is the access frequency value at time t, and α is a parameter. Based on a sensitivity analysis using our workloads, we set α to 0.4, meaning Ingens considers the page "frequently accessed" when $F_t \geq 3 \times \mathit{bitvector\ size}/4$ (i.e., 6 in our case).

We can experimentally verify the accuracy of the frequency information by checking whether pages classified as frequently accessed have their access bit set in the next scan interval: in most workloads we find the misprediction ratio to be under 3%, although random access patterns (e.g. Redis, MongoDB) can yield higher error rates depending on the dynamic request pattern.

4.2 Fast page faults

To keep the page fault handling path fast, Ingens decouples promotion decisions (policy) from huge page allocation (mechanism). The page fault handler decides when to promote a huge page and signals a background thread (called `Promote-kth`) to do the promotion (and allocation if necessary) asynchronously (Figure 3). Promote-kth compacts memory if necessary and promotes the pages identified by the page fault handler. The Ingens page fault handler never does a high-latency huge page allocation. When Promote-kth starts executing, it has a list of viable candidates for promotion; after promoting them, it resumes its scan of virtual memory to find additional candidates.

4.3 Utilization-based promotion (mitigate bloat)

Ingens explicitly and conservatively manages memory contiguity as a resource, allocating contiguous memory only when it decides a process (or VM) will use most of the allocated region based on utilization. Ingens allocates only base pages in the page fault handler and tracks base page allocations in the util bitvector. If a huge page region accumulates enough allocated base pages (90% in our prototype), the page fault handler wakes up Promote-kth to promote the base pages to a huge page.

Utilization tracking lets Ingens mitigate memory bloating. Because Ingens allocates contiguous resources only for highly utilized virtual address regions, it can control internal fragmentation. The utilization threshold provides an upper bound on memory bloat. For example, if an administrator sets the threshold to 90%, processes can use only 10% more memory in the worst case compared to a system using base pages only. The administrator can simply provision 10% additional memory to avoid unexpected swapping.

Utilization-based demotion (performance). Processes can free a base page, usually by calling `free`. If a freed base page is contained within a huge page, Linux demotes the huge page instantly. For example, Redis frees objects when deleting keys which results in a system call to free the memory. Redis uses jemalloc [20], whose `free` implementation makes an `madvise` system call with the `MADV_DONTNEED` flag to release the memory[2]. Linux demotes the huge page that contains the freed base page[3].

Demoting in-use huge pages hurts performance. Consequently, Ingens defers the demotion of high utilization huge pages. When a base page is freed within a huge page, Ingens clears the bit for the page in the util bitvector. When utilization drops below a threshold, Ingens demotes the huge page and frees the base pages whose bits are clear in the util bitvector.

4.4 Proactive batched compaction (reduce fragmentation)

Maintaining available free contiguous memory is important to satisfy large size allocation requests required when Ingens decides to promote a region to a huge page, or to satisfy other system-level contiguity in service of, for example, device drivers or user-level DMA. To this end, Ingens monitors the fragmentation state of physical memory and proactively compacts memory to reduce the latency of large contiguous allocations.

Ingens's goal is to control memory fragmentation by keeping FMFI below a threshold (that defaults to 0.8). Proactive compaction happens in Promote-kth after performing periodic scanning. Aggressive proactive compaction causes high CPU utilization, interfering with user applications. Ingens limits the maximum amount of compacted memory to 100 MB for each compaction. Compaction moves pages, which necessitates TLB invalidations. Ingens does not move frequently accessed pages to

[2]TCMalloc [35] also functions this way.

[3]Kernel version 4.5 introduces a new mechanism to free memory efficiently, called `MADV_FREE` but it also demotes huge pages instantly and causes the same memory bloating problem as `MADV_DONTNEED`.

reduce the performance impact of compaction.

4.5 Balance page sharing with performance

Ingens uses access frequency information to balance identical page sharing with application performance. It decides whether or not huge pages should be demoted to enable sharing of identical base pages contained within the huge page. In contrast to KVM, which always prioritizes memory savings over contiguity, Ingens implements a policy that avoids demoting frequently accessed huge pages. When encountering a matching identical base-page sized region within a huge page, Ingens denies sharing if that huge page is frequently accessed, otherwise it allows the huge page to be demoted for sharing.

For page sharing, the kernel marks a shared page read-only. When a process writes the page, the kernel stops sharing the page and allocates a new page to the process (similar to a copy-on-write mechanism). Ingens checks the utilization for the huge page region enclosing the new page and if it is highly utilized, it promotes the page (while Linux would wait for asychronous promotion).

4.6 Proportional promotion manages contiguity

Ingens monitors and distributes memory contiguity fairly among processes and VMs, employing techniques for proportional fair sharing of memory with an idleness penalty [81]. Each process has a share priority for memory that begins at an arbitrary but standard value (e.g, 10,000). Ingens allocates huge pages in proportion to the share value. Ingens counts infrequently accessed pages as idle memory and imposes a penalty for the idle memory. An application that has received many huge pages but is not using them actively does not get more.

We adapt ESX's adjusted shares-per-page ratio [81] to express our per-process memory promotion metric mathematically as follows.

$$\mathscr{M} = \frac{S}{H \cdot (f + \tau(1-f))} \tag{2}$$

where S is a process' (or virtual machine's or container's) huge page share priority and H is the number of bytes backed by huge pages allocated to the process. $(f + \tau(1-f))$ is a penalty factor for idle huge pages. f is the fraction of idle huge pages relative to the total number of huge pages used by this process $(0 \leq f \leq 1)$ and τ, with $0 < \tau \leq 1$, is a parameter to control the idleness penalty. Larger values of $\mathscr{M}$ receive higher priority for huge page promotion.

Intuitively, if two processes' S value are similar and one process has fewer huge pages (H is smaller), then the kernel prioritizes promotion (or allocation and promotion) of huge pages for that process. If S and H values are similar among a group of processes, the process with the largest fraction of idle pages has the smaller $\mathscr{M}$, and hence the lowest priority for obtaining new huge pages. $\tau = 1$ means $\mathscr{M}$ disregards idle memory while τ close to 0 means $\mathscr{M}$'s value is inversely proportional to the amount of idle memory.

A kernel thread (called `Scan-kth`) periodically profiles the idle fraction of huge pages in each process and updates the value of $\mathscr{M}$ for fair promotion.

4.7 Fair promotion

Promote-kth performs fair allocation of contiguity using the promotion metric. When contiguity is contended, fairness is achieved when all processes have a priority-proportional share of the available contiguity. Mathematically this is achieved by minimizing $\mathscr{O}$, defined as follows:

$$\mathscr{O} = \sum_i (\mathscr{M}_i - \bar{\mathscr{M}})^2 \tag{3}$$

The $\mathscr{M}_i$ indicates the promotion metric of process/VM i and $\bar{\mathscr{M}}$ is the mean of all process' promotion metrics. Intuitively, the formula characterizes how much process' contiguity allocation ($\mathscr{M}_i$) deviates from a fair state ($\bar{\mathscr{M}}$): in a perfectly fair state, all the $\mathscr{M}_i$ equal $\bar{\mathscr{M}}$, yielding a 0-valued $\mathscr{O}$.

In practice, to optimize $\mathscr{O}$, it suffices to iteratively select the process with the biggest $\mathscr{M}_i$, scan its address space to promote huge pages, and update $\mathscr{M}_i$ and $\mathscr{O}$. Iteration stops when $\mathscr{O}$ is close to 0 or when Promote-kth cannot generate any additional huge pages (e.g., all process are completely backed by huge pages).

An important benefit of this approach is that it does not require a performance model and it applies equally well to processes and virtual machines.

5 Implementation

Ingens is implemented in Linux 4.3.0 and contains new mechanisms to support page utilization and access frequency tracking. It also uses Linux infrastructure for huge page page table mappings and memory compaction.

5.1 Huge page promotion

Promote-kth runs as a background kernel thread and schedules huge page promotions (replacing Linux's `khugepaged`). Promote-kth maintains two priority lists: `high` and `normal`. The high priority list is a global list containing promotion requests from the page fault handler and the normal priority list is a per-application list filled in as Promote-kth periodically scans the address space. The page fault handler or a periodic timer wakes Promote-kth, which then examines the two lists and promotes in priority order.

Ingens does not reserve contiguous memory in the page fault handler. When the page fault handler requests a huge page promotion, the physical memory backing the base pages might not be contiguous. In this case, Promote-kth allocates a new 2 MB contiguous physical memory region, copies the data from the discontiguous physical memory, and maps the contiguous physical memory into the process' virtual address space. After promotion, Promote-kth frees the original discontiguous physical memory.

An application's virtual address space can grow, shrink, or be merged with other virtual address regions. These changes make new opportunities for huge page promotion which both Linux and Ingens detect by periodically scanning address spaces in the normal priority list (Linux in `khugepaged`, Ingens in Promote-kth). For example, a virtual address region that is smaller than the size of a huge page might merge with another region, allowing it to be part of a huge page.

Promote-kth compares the promotion metric (§4.6) of each application and selects the process with the highest deviation from a fair state (§4.7). It scans 16 MB of pages and sleeps for 10 seconds which is also Linux's default settings (i.e., the 1.6 MB/s in Table 4). After scanning a process' entire address space, Promote-kth records the number of promoted huge pages and if an application has too few promotions (zero in the prototype), Promote-kth excludes the application from the normal priority list for 120 seconds. This mechanism prevents an adversarial application that can monopolize Promote-kth. Such an application would have a small number of huge pages and would appear to be a good candidate to scan to increase fairness (§4.7)).

5.2 Access frequency tracking

In 2015, Linux added an access bit tracking framework [70] for version 4.3. The kernel adds an idle flag for each physical page and uses hardware access bits to track when a page remains unused. If the hardware sets an access bit, the kernel clears the idle bit. The framework provides APIs to query the idle flags and clear the access bit. Scan-kth uses this framework to find idle memory during a periodic scan of application memory. The default period is 2 seconds. Scan-kth clears the access bits at the beginning of the profiling period and queries the idle flag at the end.

In the x86 architecture, clearing the access bit causes a TLB invalidation for the corresponding page. Consequently, frequent periodic scanning can have a negative performance impact. To ameliorate this problem, Ingens supports frequency-aware profiling and sampling. When Scan-kth needs to clear the access bit of a page, it checks whether the page is frequently accessed or not. If it is not frequently accessed, Scan-kth clears the access bit, otherwise it clears it with 20% probability. Ingens uses an efficient hardware-based random number generator [18].

To verify that sampling reduces worst case overheads, we run a synthetic benchmark which reads 10 GB memory randomly without any computation, and measure the execution time for one million iterations. When Ingens resets all access bits, the execution time of the workload is degraded by 29%. Sampling-based scanning reduces the overhead to 8%. In contrast to this worst-case microbenchmark, Section 6 shows that slowdowns of Ingens on real workloads average 1%.

5.3 Limitations and future work

Linux supports transparent huge pages only for anonymous memory because huge page support for page cache pages can significantly increase I/O traffic, potentially offsetting the benefits of huge pages. If Linux adds huge pages to the page cache, it will make sense to extend Ingens to manage them with the goal of improving the read-only page cache support (implemented in FreeBSD [68]), while avoiding significant increases in I/O traffic for write-back of huge pages which are sparsely modified.

Hardware support for finer-grain tracking of access and dirty bits for huge pages would benefit Ingens. Hardware-managed access and dirty bits for all base pages within a huge page region could avoid wasted I/O on write-back of dirty pages, and enable much better informed decisions about when to demote a huge page or when huge pages can be reclaimed fairly under memory pressure.

NUMA considerations. Ingens maintains Linux's NUMA heuristics, preferring pages from a node's local NUMA region, and refusing to allocate a huge page from a different NUMA domain. All of our measurements are within a single NUMA region.

Previous work has shown that if memory is shared across NUMA nodes, huge pages may contribute to memory request imbalance across different memory controllers and reduced locality of accesses, decreasing their performance benefit [54]. This happens due to page-level false sharing, where unrelated data is accessed on the same page, and the hot page effect, which is exacerbated by the large page size. The authors propose extensions to Linux' huge page allocation mechanism to balance huge pages among NUMA domains and to split huge pages if false sharing is detected or if they become too hot. These extensions integrate nicely with Ingens. Scan-kth can already measure page access frequencies and Promote-kth can check whether huge pages need to be demoted.

6 Evaluation

We evaluate Ingens using the applications in Table 1, comparing against the performance of Linux's huge page support which is state-of-the-art. Experiments are performed on two Intel Xeon E5-2640 v3 2.60GHz CPUs (Haswell) with 64 GB memory and two 256 MB SSDs. We use Linux 4.3 and Ubuntu 14.04 for both the guest and host system. Intel supports multiple hardware page sizes of 4 KB, 2 MB and 1 GB; our experiments use only 4 KB and 2 MB huge pages. We set the number of vCPUs equal to the number of application threads.

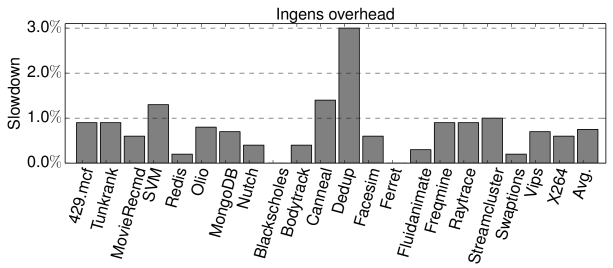


Figure 4: Performance slowdown of utilization-based promotion relative to Linux when memory is not fragmented.

Background task	**CPU utilization**
Proactive compaction	1.3%
Access bit tracking	11.4%

Table 8: CPU utilization of background tasks in Ingens. For access bit tracking, Scan-kth scans memory of MongoDB that uses 10.7GB memory.

We characterize the overheads of Ingens's basic mechanisms such as access tracking and utilization-based huge page promotion. We evaluate the performance of utilization-based promotion and demotion and Ingens ability to provide fairness across applications using huge pages. Finally, we show that Ingens's access frequency-based same page merging achieves good memory savings while preserving most of the performance benefit of huge pages. We use a single configuration to evaluate Ingens which is consistent with our examples in Sections 4 and 5: utilization threshold is 90%, Scan-kth period is 10s, access frequency tracking interval is 2 sec, and sampling ratio is 20%. Proactive batched compaction happens when FMFI is below 0.8, with an interval of 5 seconds; the maximum amount of compacted memory is 100MB; and a page is frequently accessed if $F_t \geq 6$.

6.1 Ingens overhead

Figure 4 shows the overheads introduced by Ingens for memory intensive workloads. To evaluate the performance of utilization-based huge page promotion in the unfragmented case, we run a number of benchmarks and compare their run time with Linux. Ingens's utilization-based huge page promotion slows applications down 3.0% in the worst case and 0.7% on average. The slowdowns stem primarily from Ingens not promoting huge pages as aggressively as Linux, so the workload executes with slower base pages for a short time until Ingens promotes huge pages. A secondary overhead stems from the computation of huge page utilization.

To verify that Ingens does not interfere with the performance of "normal" workloads, we measure an average performance penalty of 0.8% across the entire PARSEC 3.0 benchmark suite.

Linux	Ingens
922.3	1091.9 (1.18×)

(a) Throughput of full operation mix (requests/sec and speedup normalized to Linux).

	Event view		Homepage visit		Tag search	
	Linux	Ingens	Linux	Ingens	Linux	Ingens
Average	478	338	236	207	289	240
90th	605	354	372	226	417	299
MAX	694	649	379	385	518	507

(b) Latency (millisecond) of read-dominant operations.

Table 9: Performance result of Cloudstone WEB 2.0 Benchmark (Olio) when memory is fragmented.

Table 8 shows the CPU utilization of background tasks in Ingens. We measure the CPU utilization across 1 second intervals and take the average. For proactive compaction, we set Ingens to compact 100 MB of memory every 2 seconds (which is more aggressive than the default of 5 seconds). CPU overhead of access bit tracking depends on how many pages are scanned, so we measure the CPU utilization of Scan-kth while running MongoDB using 10.7 GB of memory.

6.2 Utilization-based promotion

To evaluate Ingens's utilization-based huge page promotion, we compare a mix of operations from the Cloudstone WEB 2.0 benchmark, which simulates a social event website. Cloudstone models a LAMP stack, consisting of a web server (nginx), PHP, and MySQL. We run Cloudstone in a KVM virtual machine and use the Rain workload generator [45] for load.

A study of the top million websites showed that in 2015 the average size exceeded 2 MB [50]. In light of this, we modify Cloudstone to serve some web pages that use about 2 MB of memory, enabling the benchmark to make better use of huge pages. The Cloudstone benchmark consists of 7 web pages, and we only modify the homepage and a page that displays social event details to use 2 MB memory. The other pages remain unchanged.

We compare throughput and latency for Cloudstone on Linux and Ingens when memory is fragmented from prior activity (FMFI = 0.9). To cause fragmentation, we run a program that allocates a large region of memory and then partially frees it.

We use Cloudstone's default operation mix: 85% read (viewing events, visiting homepage, and searching event by tag), 10% login, and 5% write (adding new events and inviting people). Our test database has 7,000 events, 2,000 people, and 900 tags. Table 9 (a) shows the throughput attained by the benchmark running on Linux and Ingens. Ingens's utilization-based promotion achieves a speedup of 1.18× over Linux. Table 9 (b) shows average and tail

Linux-nohuge	Linux	Ingens-90%	Ingens-70%	Ingens-50%
12.2 GB	20.7 GB	12.3 GB	12.9 GB	17.8 GB

(a) Redis memory consumption in different configurations. The percentage in the label is a utilization threshold.

	Throughput	90th lat.	99th lat.	99.9th lat.
Linux-nohuge	19.0K	4	5	109
Linux	21.7K	3	4	8
Ingens-90%	20.9K	3	4	64
Ingens-70%	21.1K	3	4	55
Ingens-50%	21.6K	3	4	23

(b) Redis GET Performance: Throughput (operations/sec) and latency (millisecond).

Table 10: Redis memory use and performance.

latency of the read operations in the benchmark. Ingens reduces an average latency up to 29.2% over Linux. In the tail, the reduction improves further, up to 41.4% at the 90th percentile.

Performance for Ingens improves because it reduces the average page-fault latency by not compacting memory synchronously in the page fault handler. We measure 461,383 page compactions throughout the run time of the benchmark in Linux when memory is fragmented.

When memory is not fragmented, Ingens reduces throughput by 13.4% and increases latency up to 18.1% compared with Linux. The benchmark contains many short-lived requests and Linux's greedy huge page allocation pays off by drastically reducing the total number of page faults. Ingens is less aggressive about huge page allocation to avoid memory bloat, so it incurs many more page faults.

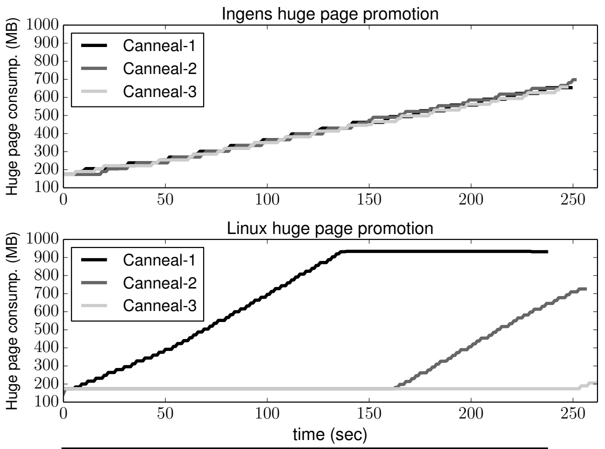
Ingens copes with this performance problem with an adaptive policy. When memory fragmentation is below 0.5 Ingens mimics Linux's aggressive huge page allocation. This policy restores Ingens's performance to Linux's levels. However, while bloat (§3.2) is not a problem for this workload, the adaptive policy increases risk of bloat in the general case. Like any management problem, it might not be possible to find a single policy that has every desirable property for a given workload. We verified that this policy performs similarly to the default policy used in Table 4, but it is most appropriate for workloads with many short-lived processes.

6.3 Memory bloating evalution

To evaluate Ingens's ability to minimize memory bloating without impacting performance, we evaluate the memory use and throughput of a benchmark using the Redis key-value store. Redis is known to be susceptible to memory bloat, as its memory allocations are often sparse. To create a sparse address space in our benchmark, we first populate Redis with 2 million keys, each with 8 KB objects and then delete 70% of the key space using a random pattern. We then measure the GET performance using the benchmark tool shipped with Redis. For Ingens, we evaluate different utilization thresholds for huge page promotion.

Table 10 shows that memory use for the 90% and 70% utilization-based configurations is very close to the case where only base pages are used. Only at 50% utilization does Ingens approach the memory use of Linux's aggressive huge page promotion.

The throughput and latency of the utilization-based approach is very close to using only huge pages. Only in the 99.9th percentile does Ingens deviate from Linux using huge pages only, while still delivering much better tail latency than Linux using base pages only.



	Canneal-1	Canneal-2	Canneal-3
Linux	181	192	195
Ingens	186	186	187

Figure 5: Huge page consumption (MB) and execution time (second). 3 instances of canneal (Parsec 3.0 benchmark) run concurrently and Promote-kth promotes huge pages. Execution time in the table excludes data loading time.

6.4 Fair huge page promotion

Ingens guarantees a fair distribution of huge pages. If applications have the same share priority (§4.6), Ingens provides the same amount of huge pages. To evaluate fairness, we run a set of three identical applications concurrently with the same share priority and idleness parameter, and measure the amount of huge pages each one holds at any point in time.

Figure 5 shows that Linux does not allocate huge pages fairly, it simply allocates huge pages to the first application that can use them (Canneal-1). In fact, Linux asynchronously promotes huge pages by scanning linearly through each application's address space, only considering the next application when it is finished with the current application. Time 160 is when Linux has pro-

Policy	Mem saving	Performance slowdown	H/M
KVM (Linux)	1438 MB (9.6%)	Tunkrank: 274 (12.7%) MovieRecmd: 210 (6.5%) SVM: 232 (20.2%)	Tunkrank: 66% MovieRecmd: 10% SVM: 72%
Huge page sharing	317 MB (2.1%)	Tunkrank: 243 MovieRecmd: 197 SVM: 193	Tunkrank: 99% MovieRecmd: 99% SVM: 99%
Ingens	1026 MB (6.8%)	Tunkrank: 247 (1.6%) MovieRecmd: 200 (1.5%) SVM: 198 (2.5%)	Tunkrank: 90% MovieRecmd: 79% SVM: 94%

Table 11: Memory saving (MB) and performance (second) trade off. H/M - huge page ratio out of total memory used. Parentheses in the Mem saving column expresses the memory saved as a percentage of the total memory (15 GB) allocated to all three virtual machines.

moted almost all of Canneal-1's address space to huge pages so only then does it begin to allocate huge pages to Canneal-2.

In contrast, Ingens promotes huge pages based on the fairness objective described in Section 4.7 and thus equally distributes the available huge pages to each application. Fair distribution of huge pages translates to fair end-to-end execution time as well. All applications finish at the same time in Ingens, while Canneal-1 finishes well before 2 and 3 on Linux.

6.5 Trade off of memory saving and performance

Finally, we evaluate the memory and performance trade-offs of identical page sharing. We run a workload mix of three different applications, each in its own virtual machine. We measure their memory use and performance slowdown under three different OS configurations: (1) KVM with aggressive page sharing, where huge pages are demoted if underlying base pages can be shared. (2) KVM where only pages of the same type may be shared and huge pages are never broken up (huge page sharing). (3) Ingens, where only infrequently used huge pages are demoted for page sharing. To avoid unused memory saving, we intentionally fit guest physical memory size to memory usages of the workloads.

Table 11 shows that KVM's aggressive page sharing saves the most memory (9.6%), but also cedes the most performance (between 6.5% and 20.2% slowdown) when compared to huge page sharing. When sharing only pages of the same type, it saves memory only 2.1%. Finally, Ingens allows us to save 6.8% of memory, while only slowing down the application up to 2.5%. The main reason for the low performance degradation is that the ratio of huge pages to total pages remains high in Ingens, due to its access frequency-based approach to huge page demotion and instant promotion when Ingens stops page sharing.

7 Related work

Virtual memory is an active research area. Our evidence of performance degradation from address translation overheads is well-corroborated [44, 53, 47, 67].

Operating system support. Navarro et al. [68] implement OS support for multiple page sizes with contiguity-awareness and fragmentation reduction as primary concerns. They propose reservation-based allocation, allocating contiguous ranges of pages in advance, and deferring promotion. Many of their ideas are widely used [80], and it forms the basis of FreeBSD's huge page support. Ingens's utilization-based promotion uses a util bitvector that is similar to the population map [68]. In contrast to that work, Ingens does not use reservation-based allocation, decouples huge page allocation from promotion decisions, and redistributes contiguity fairly when it becomes available (e.g., after process termination). Ingens has higher performance because it promotes more huge pages; it does not require promoted pages to be read-only or completely modified (§3.4). Features in modern systems such as memory compaction and same-page merging [63] pose new challenges not addressed by this previous work.

Gorman et al. [56] propose a placement policy for an OS's physical page allocator that mitigates fragmentation and promotes contiguity by grouping pages according to relocatability. Subsequent work [57] proposes a software-exposed interface for applications to explicitly request huge pages like `libhugetlbfs` [65]. The foci of Ingens, including trade-offs between memory sharing and performance, and unfair allocation of huge pages are unaddressed by previous work.

Hardware support. TLB miss overheads can be reduced by accelerating page table walks [42, 46] or reducing their frequency [52]; by reducing the number of TLB misses (e.g. through prefetching [48, 60, 74], prediction [69], or structural change to the TLB [79, 72, 71] or TLB hierarchy [47, 66, 78, 39, 38, 61, 44, 53]). Multi-page mapping techniques [79, 72, 71] map multiple pages with a single TLB entry, improving TLB reach by a small factor (e.g. to 8 or 16); much greater improvements to TLB reach are needed to deal with modern memory sizes. Direct segments [44, 53] extend standard paging with a large segment to map the majority of an address space to a contiguous physical memory region, but require application modifications and are limited to workloads able to a single large segment. Redundant memory mappings (RMM) [61] extend TLB reach by mapping *ranges* of virtually and physically contiguous pages in a range TLB. The level of additional architectural support is significant, while Ingens works on current hardware.

A number of related works propose hardware support to recover and expose contiguity. GLUE [73] groups contiguous, aligned small page translations under a single speculative huge page translation in the TLB. Speculative translations, (similar to SpecTLB [43]) can be verified by off-critical-path page-table walks, reducing effective page-table walk latency. GTSM [49] provides hardware support to leverage contiguity of physical memory extents even when pages have been retired due to bit errors. Were such features to become available, hardware mechanisms for preserving contiguity could reduce overheads induced by proactive compaction in Ingens.

Architectural assists are ultimately complementary to our own work. Hardware support can help, but higher-level coordination of hardware mechanisms by software is a fundamental necessity. Additionally, as none of these assists are likely to be realized in imminently available hardware, using techniques such as those we propose in Ingens are a *de facto* necessity.

8 Conclusion

Hardware vendors are betting on huge pages to make address translation overheads acceptable as memory capacities continue to grow. Ingens provides principled, coordinated transparent huge page support for the operating system and hypervisor, enabling challenging workloads to achieve the expected benefits of huge pages, without harming workloads that are well served by state-of the art huge page support. Ingens reduces tail-latency and bloat, while improving fairness and performance.

Acknowledgement

For their insights and comments we thank readers Mark Silberstein, Nadav Amit, Reza Taheri, Kathryn S. McKinley, the anonymous reviewers, and our shepherd Sasha Fedorova. We acknowledge funding from NSF grants CNS-1228843 and CNS-1618563.

References

[1] http://www.7-cpu.com/cpu/Skylake.html. [Accessed April, 2016].

[2] http://www.7-cpu.com/cpu/Haswell.html. [Accessed April, 2016].

[3] Apache Cloudstack. https://en.wikipedia.org/wiki/Apache_CloudStack. [Accessed April, 2016].

[4] Apache Hadoop. http://hadoop.apache.org/. [Accessed April, 2016].

[5] Apache Spark. http://spark.apache.org/docs/latest/index.html. [Accessed April, 2016].

[6] Application-friendly kernel interfaces. https://lwn.net/Articles/227818/. [March, 2007].

[7] Cloudera recommends turning off memory compaction due to high CPU utilization. http://www.cloudera.com/documentation/enterprise/latest/topics/cdh_admin_performance.html. [Accessed April, 2016].

[8] Cloudsuite. http://parsa.epfl.ch/cloudsuite/graph.html. [Accessed April, 2016].

[9] CouchBase recommends disabling huge pages. http://blog.couchbase.com/often-overlooked-linux-os-tweaks. [March, 2014].

[10] Data Plane Development Kit. http://www.dpdk.org/. [Accessed April-2016].

[11] DokuDB recommends disabling huge pages. https://www.percona.com/blog/2014/07/23/why-tokudb-hates-transparent-hugepages/. [July, 2014].

[12] Exponential moving average. https://en.wikipedia.org/wiki/Moving_average#Exponential_moving_average. [Accessed April, 2016].

[13] High CPU utilization in Hadoop due to transparent huge pages. https://www.ghostar.org/2015/02/transparent-huge-pages-on-hadoop-makes-me-sad/. [February, 2015].

[14] High CPU utilization in Mysql due to transparent huge pages. http://developer.okta.com/blog/2015/05/22/tcmalloc. [May, 2015].

[15] Huge page support in Mac OS X. `https://developer.apple.com/legacy/library/documentation/Darwin/Reference/ManPages/man2/mmap.2.html`. [Accessed April-2016].

[16] IBM cloud with KVM hypervisor. `http://www.networkworld.com/article/2230172/opensource-subnet/red-hat-s-kvm-virtualization-proves-itself-in-ibm-s-cloud.html`. [March, 2010].

[17] IBM recommends turning off huge pages due to high CPU utilization. `http://www-01.ibm.com/support/docview.wss?uid=swg21677458`. [July, 2014].

[18] Intel hardware random number generator. `https://software.intel.com/en-us/articles/intel-digital-random-number-generator-drng-software-implementation-guide`. [May, 2014].

[19] Intel HiBench. `https://github.com/intel-hadoop/HiBench/tree/master/workloads`. [Accessed April, 2016].

[20] Jemalloc. `http://www.canonware.com/jemalloc/`. [Accessed April-2016].

[21] Large-page support in Windows. `https://msdn.microsoft.com/en-us/library/windows/desktop/aa366720(v=vs.85).aspx`. [Accessed April-2016].

[22] Liblinear. `https://www.csie.ntu.edu.tw/~cjlin/liblinear/`. [Accessed April, 2016].

[23] MongoDB. `https://www.mongodb.com/`. [Accessed April, 2016].

[24] MongoDB recommends disabling huge pages. `https://docs.mongodb.org/manual/tutorial/transparent-huge-pages/`. [Accessed April, 2016].

[25] Movie recommendation with Spark. `http://ampcamp.berkeley.edu/big-data-mini-course/movie-recommendation-with-mllib.html`. [Accessed April, 2016].

[26] NuoDB recommends disabling huge pages. `http://www.nuodb.com/techblog/linux-transparent-huge-pages-jemalloc-and-nuodb`. [May, 2014].

[27] OpenStack. `https://openvirtualizationalliance.org/what-kvm/openstack`. [Accessed April-2016].

[28] PARSEC 3.0 benchmark suite. `http://parsec.cs.princeton.edu/`. [Accessed April, 2016].

[29] Redis. `http://redis.io/`. [Accessed April, 2016].

[30] Redis recommends disabling huge pages. `http://redis.io/topics/latency`. [Accessed April, 2016].

[31] Redis SSD swap discussion. `http://antirez.com/news/52`. [March, 2013].

[32] SAP IQ recommends disabling huge pages. `http://scn.sap.com/people/markmumy/blog/2014/05/22/sap-iq-and-linux-hugepagestransparent-hugepages`. [May, 2014].

[33] SPEC CPU 2006. `https://www.spec.org/cpu2006/`. [Accessed April, 2016].

[34] Splunk recommends disabling huge pages. `http://docs.splunk.com/Documentation/Splunk/6.1.3/ReleaseNotes/SplunkandTHP`. [December, 2013].

[35] Thread-caching malloc. `http://goog-perftools.sourceforge.net/doc/tcmalloc.html`. [Accessed April-2016].

[36] Transparent huge pages in 2.6.38. `https://lwn.net/Articles/423584/`. [January, 2011].

[37] VoltDB recommends disabling huge pages. `https://docs.voltdb.com/AdminGuide/adminmemmgt.php`. [Accessed April, 2016].

[38] Jeongseob Ahn, Seongwook Jin, and Jaehyuk Huh. Revisiting hardware-assisted page walks for virtualized systems. In *International Symposium on Computer Architecture (ISCA)*, 2012.

[39] Jeongseob Ahn, Seongwook Jin, and Jaehyuk Huh. Fast two-level address translation for virtualized systems. In *IEEE Transactions on Computers*, 2015.

[40] AMD. *AMD-V Nested Paging*, 2010. `http://developer.amd.com/wordpress/media/2012/10/NPT-WP-1%201-final-TM.pdf`.

[41] Jean Araujo, Rubens Matos, Paulo Maciel, Rivalino Matias, and Ibrahim Beicker. Experimental evaluation of software aging effects on the eucalyptus cloud computing infrastructure. In *Middleware Industry Track Workshop*, 2011.

[42] Thomas W. Barr, Alan L. Cox, and Scott Rixner. Translation caching: Skip, don't walk (the page table). In *International Symposium on Computer Architecture (ISCA)*, 2010.

[43] Thomas W. Barr, Alan L. Cox, and Scott Rixner. Spectlb: A mechanism for speculative address translation. In *International Symposium on Computer Architecture (ISCA)*, 2011.

[44] Arkapravu Basu, Jayneel Gandhi, Jichuan Chang, Mark D. Hill, and Michael M. Swift. Efficient virtual memory for big memory servers. In *International Symposium on Computer Architecture (ISCA)*, 2013.

[45] Aaron Beitch, Brandon Liu, Timothy Yung, Rean Griffith, Armando Fox, and David Patterson. Rain: A workload generation toolkit for cloud computing applications. In *U.C. Berkeley Technical Publications (UCB/EECS-2010-14)*, 2010.

[46] Abhishek Bhattacharjee. Large-reach memory management unit caches. In *International Symposium on Microarchitecture*, 2013.

[47] Abhishek Bhattacharjee, Daniel Lustig, and Margaret Martonosi. Shared last-level TLBs for chip multiprocessors. In *IEEE International Symposium on High Performance Computer Architecture (HPCA)*, 2011.

[48] Abhishek Bhattacharjee and Margaret Martonosi. Characterizing the TLB behavior of emerging parallel workloads on chip multiprocessors. In *International Conference on Parallel Architectures and Compilation Techniques (PACT)*, 2009.

[49] Yu Du, Miao Zhou, B.R. Childers, D. Mosse, and R. Melhem. Supporting superpages in non-contiguous physical memory. In *IEEE International Symposium on High Performance Computer Architecture (HPCA)*, 2015.

[50] Tammy Everts. The average web page is more than 2 MB size. `https://www.soasta.com/blog/page-bloat-average-web-page-2-mb/`. [June, 2015].

[51] Michael Ferdman, Almutaz Adileh, Onur Kocberber, Stavros Volos, Mohammad Alisafaee, Djordje Jevdjic, Cansu Kaynak, Adrian Daniel Popescu, Anastasia Ailamaki, and Babak Falsafi. Clearing the clouds: A study of emerging scale-out workloads on modern hardware. In *Proceedings of the Seventeenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XVII, pages 37–48, New York, NY, USA, 2012. ACM.

[52] Jayneel Gandhi, , Mark D. Hill, and Michael M. Swift. Exceeding the best of nested and shadow paging. In *International Symposium on Computer Architecture (ISCA)*, 2016.

[53] Jayneel Gandhi, Arkaprava Basu, Mark D. Hill, and Michael M. Swift. Efficient memory virtualization. In *International Symposium on Microarchitecture*, 2014.

[54] Fabien Gaud, Baptiste Lepers, Jeremie Decouchant, Justin Funston, Alexandra Fedorova, and Vivien Quéma. Large pages may be harmful on numa systems. In *Proceedings of the 2014 USENIX Conference on USENIX Annual Technical Conference*, USENIX ATC'14, pages 231–242, Berkeley, CA, USA, 2014. USENIX Association.

[55] Joseph E. Gonzalez, Yucheng Low, Haijie Gu, Danny Bickson, and Carlos Guestrin. Powergraph: Distributed graph-parallel computation on natural graphs. In *Presented as part of the 10th USENIX Symposium on Operating Systems Design and Implementation (OSDI 12)*, pages 17–30, Hollywood, CA, 2012. USENIX.

[56] Mel Gorman and Patrick Healy. Supporting superpage allocation without additional hardware support. In *Proceedings of the 7th International Symposium on Memory Management*, 2008.

[57] Mel Gorman and Patrick Healy. Performance characteristics of explicit superpage support. In *Workshorp on the Interaction between Operating Systems and Computer Architecture (WIOSCA)*, 2010.

[58] Mel Gorman and Andy Whitcroft. The what, the why and the where to of anti-fragmentation. In *Linux Symposium*, 2005.

[59] Intel Corporation. *Intel 64 and IA-32 Architectures Software Developers Manual*, 2016. `https://www-ssl.intel.com/content/dam/www/public/us/en/documents/manuals/64-ia-32-architectures-software-developer-manual-325462.pdf`.

[60] Gokul B. Kandiraju and Anand Sivasubramaniam. Going the distance for TLB prefetching: An application-driven study. In *International Symposium on Computer Architecture (ISCA)*, 2002.

[61] Vasileios Karakostas, Jayneel Gandhi, Furkan Ayar, Adrin Cristal, Mark D. Hill, Kathryn S. McKinley, Mario Nemirovsky, Michael M. Swift, and Osman nsal. Redundant memory mappings for fast access to large memories. In *International Symposium on Computer Architecture (ISCA)*, 2015.

[62] Avi Kivity, Yaniv Kamay, Dor Laor, Uri Lublin, and Anthony Liguori. KVM: The linux virtual machine monitor. In *Linux Symposium*, 2007.

[63] Kernel Same-page Merging. `https://en.wikipedia.org/wiki/Kernel_same-page_merging`. [Accessed April, 2016].

[64] Ching-Pei Lee and Chih-Jen Lin. Large-scale linear RankSVM. *Neural Comput.*, 26(4):781–817, April 2014.

[65] Huge Pages Part 2 (Interfaces). `https://lwn.net/Articles/375096/`. [February, 2010].

[66] Daniel Lustig, Abhishek Bhattacharjee, and Margaret Martonosi. TLB improvements for chip multiprocessors: Inter-core cooperative prefetchers and shared last-level TLBs. *ACM Transactions on Architecture and Code Optimization (TACO)*, 2013.

[67] Timothy Merrifield and H. Reza Taheri. Performance implications of extended page tables on virtualized x86 processors. In *Proceedings of the12th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*, VEE '16, pages 25–35, New York, NY, USA, 2016. ACM.

[68] Juan Navarro, Sitaram Iyer, Peter Druschel, and Alan Cox. Practical, transparent operating system support for superpages. In *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2002.

[69] M.-M. Papadopoulou, Xin Tong, A. Seznec, and A. Moshovos. Prediction-based superpage-friendly TLB designs. In *IEEE International Symposium on High Performance Computer Architecture (HPCA)*, 2015.

[70] Idle Page Tracking. `http://lxr.free-electrons.com/source/Documentation/vm/idle_page_tracking.txt`. [November, 2015].

[71] Binh Pham, Abhishek Bhattacharjee, Yasuko Eckert, and Gabriel H. Loh. Increasing TLB reach by exploiting clustering in page translations. In *IEEE International Symposium on High Performance Computer Architecture (HPCA)*, 2014.

[72] Binh Pham, Viswanathan Vaidyanathan, Aamer Jaleel, and Abhishek Bhattacharjee. CoLT: Coalesced large-reach TLBs. In *International Symposium on Microarchitecture*, 2012.

[73] Binh Pham, Jan Vesely, Gabriel Loh, and Abhishek Bhattacharjee. Large pages and lightweight memory management in virtualized systems: Can you have it both ways? In *International Symposium on Microarchitecture*, 2015.

[74] Ashley Saulsbury, Fredrik Dahlgren, and Per Stenström. Recency-based TLB preloading. In *International Symposium on Computer Architecture (ISCA)*, 2000.

[75] Tom Shanley. *Pentium Pro Processor System Architecture*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1st edition, 1996.

[76] Richard L. Sites and Richard T. Witek. *ALPHA architecture reference manual*. Digital Press, Boston, Oxford, Melbourne, 1998.

[77] Will Sobel, Shanti Subramanyam, Akara Sucharitakul, Jimmy Nguyen, Hubert Wong, Arthur Klepchukov, Sheetal Patil, O Fox, and David Patterson. Cloudstone: Multi-platform, multi-language benchmark and measurement tools for web 2.0, 2008.

[78] Shekhar Srikantaiah and Mahmut Kandemir. Synergistic tlbs for high performance address translation in chip multiprocessors. In *International Symposium on Microarchitecture*, 2010.

[79] M. Talluri and M. D. Hill. Surpassing the TLB performance of superpages with less operating system support. In *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 1994.

[80] Transparent Hugepages. `https://lwn.net/Articles/359158/`. [October, 2009].

[81] Carl A. Waldspurger. Memory resource management in VMware ESX server. In *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2002.

Diamond: Automating Data Management and Storage for Wide-area, Reactive Applications

Irene Zhang Niel Lebeck Pedro Fonseca Brandon Holt Raymond Cheng
Ariadna Norberg Arvind Krishnamurthy Henry M. Levy

University of Washington

Abstract

Users of today's popular wide-area apps (e.g., Twitter, Google Docs, and Words with Friends) must no longer save and reload when updating shared data; instead, these applications are *reactive*, providing the illusion of continuous synchronization across mobile devices and the cloud. Achieving this illusion poses a complex *distributed data management* problem for programmers. This paper presents the first *reactive data management service*, called Diamond, which provides persistent cloud storage, reliable synchronization between storage and mobile devices, and automated execution of application code in response to shared data updates. We demonstrate that Diamond greatly simplifies the design of reactive applications, strengthens distributed data sharing guarantees, and supports automated reactivity with low performance overhead.

1 Introduction

The modern world's ubiquitous mobile devices, infinite cloud storage, and nearly constant network connectivity are changing applications. Led by social networks (e.g., Twitter), social games (e.g., Words with Friends) and collaboration tools (e.g., Google Docs), today's popular applications are *reactive* [41]: they provide users with the illusion of *continuous synchronization* across their devices without requiring them to explicitly save, reload, and exchange shared data. This trend, not limited merely to mobile apps, includes the latest distributed versions of traditional desktop apps on both Windows [13] and OSX [4].

Maintaining this illusion presents a challenging *distributed data management* problem for application programmers. Modern reactive applications consist of *widely distributed* processes sharing data across mobile devices, desktops, and cloud servers. These processes make concurrent data updates, can stop or fail at any time, and may be connected by slow or unreliable links. While distributed storage systems [17, 77, 15, 23, 20] provide persistence and availability, programmers still face the formidable challenge of synchronizing updates between application processes and distributed storage in a *fault-tolerant*, *consistent* manner.

This paper presents *Diamond*, the first *reactive data management service* (RDS) for wide-area applications that continuously synchronizes shared application data across distributed processes. Specifically, Diamond performs the following functions on behalf of an application: (1) it ensures that updates to shared data are consistent and durable, (2) it reliably coordinates and synchronizes shared data updates across processes, and (3) it automatically triggers *reactive code* when shared data changes so that processes can perform appropriate tasks. For example, when a user updates data on one device (e.g., a move in a multi-player game), Diamond persists the update, reliably propagates it to other users' devices, and transparently triggers application code on those devices to react to the changes.

Reactive data management in the wide-area context requires a balanced consideration of performance trade-offs and reasoning about complex correctness requirements in the face of concurrency. Diamond implements the difficult mechanisms required by these applications (such as logging and concurrency control), letting programmers focus on high-level data-sharing requirements (e.g., atomicity, concurrency, and data layout). Diamond introduces three new concepts:

1. **Reactive Data Map** (`rmap`), a primitive that lets applications create *reactive data types* – shared, persistent data structures – and map them into the Diamond data management service so it can automatically synchronize them across distributed processes and persistent storage.
2. **Reactive Transactions**, an interactive transaction type that automatically *re-executes* in response to shared data updates. These "live" transactions run *application code* to make local, application-specific updates (e.g., UI changes).
3. **Data-type Optimistic Concurrency Control** (DOCC), a mechanism that leverages data-type semantics to concurrently commit transactions executing commutative operations (e.g., writes to different list elements, increments to a counter). Our experiments show that DOCC copes with wide-area

latencies very effectively, reducing abort rates by up to 5x.

We designed and implemented a Diamond prototype in C++ with language bindings for C++, Python, and Java on both x86 and Android platforms. We evaluate Diamond by building and measuring both Diamond and custom versions (using explicit data management) of four reactive apps. Our experiments show that Diamond significantly reduces the complexity and size of reactive applications, provides strong transactional guarantees that eliminate data races, and supports automatic reactivity with performance close to that of custom-written reactive apps.

2 Traditional Data Management Techniques for Reactive Apps

Reactive applications require synchronized access to distributed shared data, similar to shared virtual memory systems [46, 10]. For practical performance in the wide-area environment, apps must be able to control: (1) *what* data in each process is shared, (2) *how* often it is synchronized, and (3) *when* concurrency control is needed. Existing applications use one of several approaches to achieve synchronization with control. This section demonstrates that these approaches are all complex, error-prone, and make it difficult to reason about application data consistency.

As an example, we analyze a simple social game based on the 100 game [1]. Such games are played by millions [78], and their popularity changes constantly; therefore, game developers want to build them quickly and focus on game logic rather than data management. Because game play increasingly uses real money (almost $2 billion last year [24]), their design parallels other reactive applications where correctness is crucial (e.g., apps for first responders [52] and payment apps [81, 72]).

In the 100 game, players alternately add a number between 1 and 10 to the current sum, and the first to reach 100 wins. Players make moves and can join or leave the game at different times; application processes can fail at any time. Thus, for safety, the game must maintain traditional ACID guarantees – atomicity, consistency, isolation and durability – as well as *reactivity* for data updates. We call this combination of properties ACID+R. While a storage system provides ACID guarantees for its own data, those guarantees *do not extend to application processes*. In particular, pushing updates to storage on mobile devices is insufficient for reactivity because application processes must re-compute local data *derived* from shared data to make changes *visible* to users and other components.

2.1 Roll-your-own Data Management

Many current reactive apps "roll-their-own" *application-specific* synchronization across distributed processes *on top of* general-purpose distributed storage (e.g., Spanner [17], Dropbox [23]). Figure 1

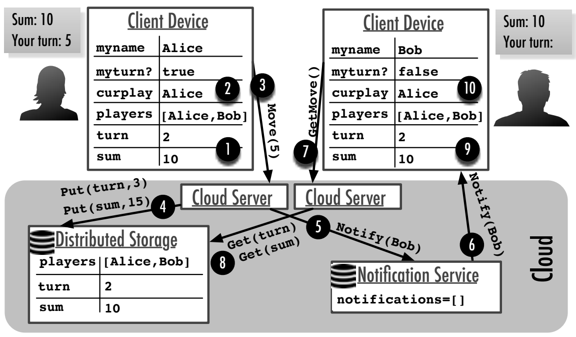


Figure 1: **The 100 game.** Each box is a separate address space. `players`, `turn` and `sum` are shared across address spaces and the storage system; `myturn?` and `curplay` are derived from shared data. When shared values change, the app manually updates distributed storage, other processes with the shared data, and any data in those processes derived from shared data, as shown by the numbered steps needed to propagate Alice's move to Bob.

shows a typical three-tiered architecture used by these apps (e.g., PlayFish uses it to serve over 50 million users/month [34]). Processes on *client devices* access stateless *cloud servers*, which store persistent game state in a *distributed storage* system and use a reliable *notification service* (e.g., Thialfi [3]) to trigger changes in other processes for reactivity. While all application processes can fail, we assume strong guarantees – such as durability and linearizability – for the storage system and notification service. Although such apps could rely on a single server to run the game, this would create a centralized failure point. Clients cache game data to give users a responsive experience and to reduce load on the cloud servers [34].

The numbers in Figure 1 show the data management steps that the application must explicitly perform for Alice's move (adding 5 to the sum). Alice's client: (1) updates `turn` and `sum` locally, (2) calculates new values for `myturn?` and `curplay`, and (3) sends the move to a cloud server. The server: (4) writes `turn` and `sum` to distributed storage, and (5) sends a notification to Bob. The notification service: (6) delivers the notification to Bob's client, which (7) contacts a cloud server to get the latest move. The server: (8) reads from distributed storage and returns the latest `turn` and `sum`. Bob's client: (9) updates `turn` and `sum` locally, and (10) re-calculates `myturn?` and `curplay`.

Note that such data management must be customized to such games, making it difficult to implement a general-purpose solution. For example, only the application knows that: (1) clients share `turn` and `sum` (but not `myname`), (2) it needs to synchronize `turn` and `sum` after each turn (but not `players`), and (3) it does not need concurrency control because `turn` already coordinates moves.

Correctly managing this application data demands that the programmer reason about failures and data races at every step. For example, the cloud server could fail in the middle of step 4, violating atomicity. It could also fail between steps 4 and 5, making the application appear as if it is no longer reactive.

A new player, Charlie, could join the game while Bob makes his move, leading to a race; if Alice receives Bob's notification first, but Charlie writes to storage first, then both Alice and Charlie would think that it was their turn, violating isolation.

Finally, even if the programmer were to correctly handle every failure and data race *and* write bug-free code, reasoning about the consistency of application data would prove difficult. Enforcing a single global ordering of join, leave and move operations requires application processes to either forgo caching shared data (or data derived from shared data) altogether or invalidate all cached copies and update the storage system atomically on every operation. The first option is not realistic in a wide-area environment, while the second is not possible when clients may be unreachable.

2.2 Wide-area Storage Systems

A simple, alternative way to manage data manually is to store shared application data in a wide-area storage system (e.g., Dropbox [23]). That is, rather than calling `move` in step 3, the application stores and updates `turn` and `sum` in a wide-area storage system. Though simple, this design can be very expensive. Distributed file systems are not designed to frequently synchronize small pieces of data, so their coarse granularity can lead to moving more data than necessary and false sharing.

Further, while this solution synchronizes Alice's updates with the cloud, it does not ensure that Bob receives Alice's updates. To simulate reactive behavior and ensure that Bob sees Alice's updates, Alice must still use a wide-area notification system (e.g., Apple Push Notifications [6]) to notify Bob's client after her update. Unfortunately, this introduces a race condition: if Bob's client receives the notification before the wide-area storage system synchronizes Alice's update, then Bob will not see Alice's changes. Worse, Bob will never check the storage system again, so he will never see Alice's update, leaving him unable to make progress. Thus, this solution retains all of the race conditions described in Section 2.1 *and* introduces some new ones.

2.3 Reactive Programming Frameworks

Several programming frameworks (e.g., Firebase [26], Parse [60] with React [64], Meteor [51]) have recently been commercially developed for reactive applications. These frameworks combine storage and notification systems and automate data management and synchronization across systems. However, they do not provide a clear consistency model, making it difficult for programmers to reason about the guarantees provided by their synchronization mechanisms. Further, they offer no distributed concurrency control, leaving application programmers to contend with race conditions; for example, they can lead to the race condition described in Section 2.1.

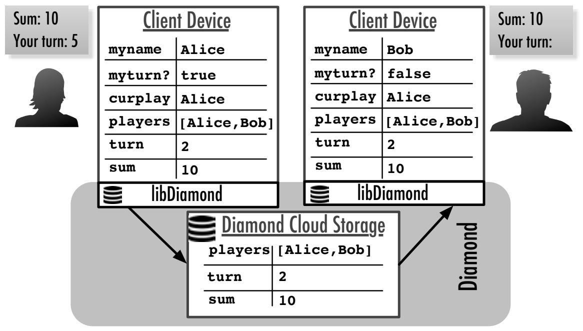


Figure 2: **Diamond 100 game data model.** The app `rmaps` `players`, `turn` and `sum`, updates them in read-write transactions and computes `myturn?` and `curplay` in a reactive transaction.

3 Diamond's System and Programming Model

Diamond is a new programming platform designed to simplify the development of wide-area reactive applications. This section specifies its data and transaction models and system call API.

3.1 System Model

Diamond applications consist of processes running on mobile devices and cloud servers. Processes can communicate through Diamond or over a network, which can vary from local IPC to the Internet. Every application process is linked to a client-side library, called LIBDIAMOND, which provides access to the shared *Diamond cloud* – a highly available, fault-tolerant, durable storage system. Diamond subsumes some applications' server-side functionality, but our goal is not to eliminate such code. We expect cloud servers to continue providing reliable and efficient access to computation and datacenter services (e.g., data mining) while accessing shared data needed for these tasks through Diamond.

Figure 2 shows the 100 game data model using Diamond. Compared to Figure 1, the application can directly read and write to shared data in memory, and Diamond ensures updates are propagated to cloud storage and other processes. Further, Diamond's strong transactional guarantees eliminate the need for programmers to reason about failures and concurrency.

3.2 Data Model

Diamond supports *reactive data types* for fine-grained synchronization, efficient concurrency control, and persistence. As with popular data structure stores [19], such as Redis [67] and Riak [68], we found that simple data types are general enough to support a wide range of applications

Table 1: **Reactive data types.**

Type	Operations	Description
`Boolean`	`Get()`, `Put(bool)`	Primitive boolean
`Long`	`Get()`, `Put(long)`	Primitive number
`String`	`Get()`, `Put(str)`	Primitive string
`Counter`	`Get()`, `Put(long)` `Increment(long)` `Decrement(long)`	Long Counter
`IDGen`	`GetUID()`	Unique ID generator
`LongSet`	`Get(idx)`, `Contains(long)` `Insert(long)`	Ordered number set
`LongList`	`Get(idx)`, `Set(idx, long)` `Append(long)`	Number list
`StringSet`	`Get(idx)`, `Contains(str)` `Insert(str)`	Ordered string set
`StringList`	`Get(idx)`, `Set(idx, str)` `Append(str)`	String list
`HashTable`	`Get(key)`, `Set(key, val)`	Unordered map

and provide the necessary semantics to enable commutativity and avoid false sharing. Table 1 lists the supported persistent data types and their operations. In addition to primitive data types, like `String`, Diamond supports simple Conflict-free Replicated Data-types (CRDTs) [69] (e.g., `Counter`) and collection types (e.g., `LongSet`) with efficient type-specific interfaces. Using the most specific type possible provides the best performance (e.g., using a `Counter` for records that are frequently incremented).

A single Diamond *instance* provides a set of *tables*; each table is a key-to-data-type map, where each entry, or *record*, has a single persistent data type. Applications access Diamond through language bindings; however, applications need not be written in a single language. We currently support C++, Python and Java on both x86 and Android but could easily add support for other languages (e.g., Swift [5]).

3.3 System Calls

While apps interact with Diamond largely through reactive data types, we provide a minimal system call interface, shown in Table 2, to support transactions and `rmap`.

3.3.1 The `rmap` Primitive

`rmap` is Diamond's key abstraction for providing shared memory that is flexible, persistent, and reactive across wide-area application processes. Applications call `rmap` with an application variable and a key to the Diamond record, giving them control over what data in their address space is shared and how it is organized. In this way, different application processes (e.g., an iOS and an Android client) and different application versions (e.g., a new and current code release) can effectively share data. When `rmap`ping records to variables, the data types must match. Diamond's system call library checks at runtime and returns an error from the `rmap` call if a mismatch occurs.

Table 2: **Diamond system calls.**

System call	Description
`create(table, [isolation])`	Create table
`status = rmap(var, table, key)`	Bind `var` to key
`id = execute_txn(func, cb)`	Start read-write transaction
`id = register_reactxn(func)`	Start reactive transaction
`reactxn_stop(txn_id)`	Stop re-executing
`commit_txn()`, `abort_txn()`	Commit/Abort and exit

3.3.2 Transaction model

Application processes use Diamond transactions to read and write `rmap`ped variables. Diamond transactions are *interactive* [73], i.e., they let applications interleave application code with accesses to reactive data types. We support both standard *read-write transactions* and new *reactive transactions*. Applications cannot execute transactions across `rmap`ped variables from different tables, while operations executed outside transactions are treated as single-op transactions.

Read-write transactions. Diamond's read-write transactions let programmers safely and easily access shared reactive data types despite failures and concurrency. Applications invoke read-write transactions using `execute_txn`. The application passes closures for both the transaction and a completion callback. Within the transaction closure, the application can read or write `rmap`ped variables and variables in the closure, but it cannot modify program variables outside the closure. This limitation ensures: (1) the transaction can access all needed variables when it executes asynchronously (and they have not changed), and (2) the application is not affected by the side effects of aborted transactions. Writes to `rmap`ped variables are buffered locally until commit, while reads go to the client-side cache or to cloud storage.

Before `execute_txn` returns, Diamond logs the transaction, with its read and write sets, to persistent storage. This step guarantees that the transaction will eventually execute and that the completion callback will eventually execute even if the client crashes and restarts. This guarantee lets applications buffer transactions if the network is unavailable and easily implement custom retry functionality in the completion callback. If the callback reports that the transaction successfully committed, then Diamond guarantees ACID+R semantics for all accesses to `rmap`ped records; we discuss these in more detail in Section 3.4. On abort, Diamond rolls back all local modifications to `rmap`ped variables.

Reactive transactions. Reactive transactions help application processes automatically propagate changes made to reactive data types. Each time a read-write transaction modifies an `rmap`ped variable in a reactive transaction's read set, the reactive transaction re-executes, propagating changes to derived local variables. As a result, reactive transactions provide a "live" view that gives the illusion of reactivity while maintaining an imperative programming style comfortable to application programmers. Further, because they read a consistent snapshot of `rmapped` data, reactive transactions avoid application-level bugs common to reactive programming models [48].

Applications do not explicitly invoke reactive transactions; instead, they register them by passing a closure to `register_reactxn`, which returns a `txn_id` that can be used to unregister the transaction with `reactxn_stop`. Within the reactive transaction closure, the application can read but not write `rmapped` records, preventing potential data flow cycles. Since reactive transactions are designed to propagate changes to local variables, the application can read and write to local variables at any time and trigger side-effects (i.e., print-outs, updating the UI). Diamond guarantees that reactive transactions never abort because it commits read-only transactions locally at the client. Section 4 details the protocol for reactive transactions.

Reactive transactions run in a background thread, concurrently with application threads. Diamond transactions do not protect accesses to local variables, so the programmer must synchronize with locks or other mechanisms. The read set of a reactive transaction can change on every execution; Diamond tracks the read set from the latest execution. Section 6.2 explains how to use reactive transactions to build general-purpose, reactive UI elements.

3.4 Reactive Data Management Guarantees

Diamond's guarantees were designed to meet the requirements of reactive applications specified in Section 2, eliminating the need for each application to implement its own complex data management. To do so, Diamond enforces *ACID+R* guarantees for reactive data types:

- **Atomicity:** All or no updates to shared records in a read-write transaction succeed.
- **Consistency:** Accesses in all transactions reflect a consistent view of shared records.[1]
- **Isolation:** Accesses in all transactions reflect a global ordering of committed read-write transactions.
- **Durability:** Updates to shared records in committed read-write transactions are never lost.
- **Reactivity:** Accesses to modified records in registered reactive transactions will eventually re-execute.

These guarantees create a *producer-consumer* relationship: Diamond's read-write transactions *produce* updates to reactive data types, while reactive transactions *consume* those updates and propagate them to locally derived data. However, unlike the traditional producer-consumer

[1]The C in ACID is not well defined outside a database context. Diamond simply guarantees that each transaction reads a consistent snapshot.

Table 3: **Diamond's isolation levels.** Isolation levels for read-write transactions and associated ones for reactive transactions.

Stronger Guarantees
Fewer Aborts

Read-write Isolation Level	Reactive Isolation Level
Strict Serializability	Serializable Snapshot
Snapshot Isolation	Serializable Snapshot
Read Committed	Read Committed

paradigm, this mechanism is *transparent* to applications because the ACID+R guarantees ensure that Diamond *automatically* re-executes the appropriate reactive transactions when read-write transactions commit.

Table 3 lists Diamond's isolation levels, which can be set per table. Diamond's default is strict serializability because it eliminates the need for application programmers to deal with inconsistencies caused by data races and failures. Lowering the isolation level leads to fewer aborts and more concurrency; however, more anomalies arise, so applications should either expect few conflicts, require offline access, or tolerate inaccuracy (e.g., Twitter's most popular hash tag statistics). Section 5.1 describes how DOCC increases concurrency and reduces aborts for transactions even at the highest isolation levels.

3.5 A Simple Code Example

To demonstrate the power of Diamond to simplify reactive applications, Figure 3 shows code to implement the 100 game from Section 2 in Diamond. This implementation provides persistence, atomicity, isolation and reactivity for every join and move operation in only 34 lines of code. We use three reactive data types for shared game data, declared on line 2 and `rmapped` in lines 7-9. It is important to ensure a strict ordering of updates, so we create a table in strict serializable mode on line 6. On line 12, we define a general-purpose transaction callback for potential transaction failures. On line 16, we execute a read-write transaction to add the player to the game, passing `myname` by value into the transaction closure. Using DOCC allows Diamond to commit two concurrent executions of this transaction while guaranteeing strict serializability.

Line 20 registers a reactive transaction to print out the score and current turn. Diamond's ACID+R guarantees ensure that the transaction re-executes if `players`, `turn` or `sum` change, so the user always has a consistent, up-to-date view. Note that we can print to stdout because the reactive transaction will not abort, and the printouts reflect a serializable snapshot, avoiding reactive glitches [48]. On line 32, we wait for user input in the `while` loop and use a read-write transaction to commit the entered move.

Diamond's strong guarantees eliminate the need for programmers to reason about data races or failures. Taking our examples from Section 2, Diamond ensures that when the game commits Alice's move, the move is never

```
int main(int argc, char **argv) {
   DStringSet players; DCounter sum, turn;
   string myname = string(argv[1]);

   // Map game state
   create("100game", STRICT_SERIALIZABLE);
   rmap(players, "100game", "players");
   rmap(sum, "100game", "sum");
   rmap(turn, "100game", "turn");

   // General-purpose callback, exit if txn failed
   auto cb = [] (txn_func_t txn, int status) {
      if (status == REPLY_FAIL) exit(1); };

   // Add user to the game
   execute_txn([myname] () {
      players.Insert(myname); }, cb);

   // Set up our print outs
   register_reactxn([myname] () {
      string curplay =
         players[turn % players.size()];
      bool myturn = myname == curplay;
      cout << "Sum: " << sum << "\n";
      if (sum >= 100)
         cout << curplay << " won!";
      else if (myturn)
         cout << "Your turn: ";
   });

   // Cycle on user input
   while (1) {
      int inc; cin >> inc;
      execute_txn([myname, inc] () {
         bool myturn =
            myname == players[turn % players.size()];
         // check inputs
         if (!myturn || inc < 1 || inc > 10) {
            abort_txn(); return;
         }
         sum += inc; if (sum < 100) turn++;
      }, cb);
   }
   return 0;
}
```

Figure 3: **Diamond code example.** Implementation of the 100 game using Diamond. Omitting includes, set up, and error handling, this code implements a working, C++ version of the 100 game [1]. `DStringSet`, `DLong` and `DCounter` are reactive data types provided by the Diamond C++ library.

lost and Bob eventually sees it. Diamond also ensures that, if Charlie joins before Bob makes his move, Alice either sees Charlie join without Bob's move, or both, but never sees Bob's move without seeing Charlie join. As a result, programmers no longer need to reason about race conditions, greatly simplifying the game's design. To our knowledge, no other system provides all of Diamond's ACID+R properties.

3.6 Offline Support

Wi-Fi and cellular data networks have become widely available, and reactive applications typically have limited offline functionality; thus, Diamond focuses on providing *online reactivity*, unlike storage systems (e.g., Bayou [77] and Simba [61]). However, Diamond still provides limited offline support. If the network is unavailable, `execute_txn` logs and transparently retries, while Diamond's CRDTs make it more likely that transactions commit after being retried. For applications with higher contention, Diamond's read committed mode enables commits locally at the client while offline, and any modifications eventually converge to a consistent state for Diamond's CRDTs.

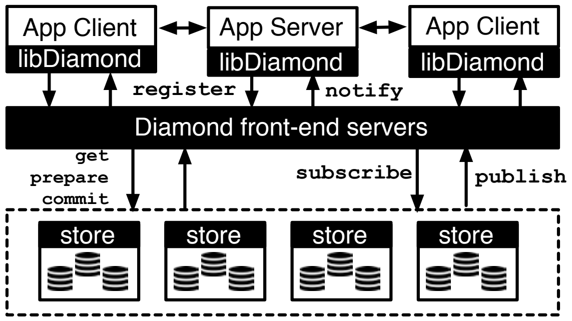


Figure 4: **Diamond architecture.** Distributed processes share a single instance of the Diamond storage system.

3.7 Security

Similar to existing client-focused services, like Firebase [26] and Dropbox [23], Diamond trusts application clients not to be malicious. Application clients authenticate with the Diamond cloud through their LIBDIAMOND client before they can `rmap` or access reactive data types. Diamond supports isolation between users through *access control lists* (ACLs); applications can set `rmap`, read, and write permissions per table. Within tables, keys function as capabilities; a client with a key to a record has permission to access it. Applications can defend against potentially malicious clients by implementing server-side security checks using reactive transactions on a secure cloud server.

4 Diamond's System Design

This section relates Diamond's architecture, the design of `rmap`, and its transaction protocols.

4.1 Data Management Architecture

Figure 4 presents an overview of Diamond's key components. Each LIBDIAMOND client provides client-side caching and access to cloud storage for the application process. It also registers, tracks and re-executes reactive transactions and keeps a persistent transaction log to handle device and network failures.

The Diamond cloud consists of *front-end servers* and *back-end storage* servers, which together provide durable storage and reliable notifications for reactive transactions. Front-end servers are scalable, stateless nodes that provide LIBDIAMOND clients access to Diamond's back-end storage, which is partitioned for scalability and replicated (using Viewstamped Replication (VR) [58]) for fault tolerance. LIBDIAMOND clients could directly access back-end storage, but front-end servers give clients a single connection point to the Diamond cloud, avoiding the need for them to authenticate with many back-end servers or track the partitioning scheme.

4.2 `rmap` and Language Bindings

Diamond language bindings implement the library of reactive data types for apps to use as `rmap` variables. Diamond *interposes* on every operation to an `rmap`ped variable. During a transaction, LibDiamond collects an *operation set* for DOCC to later check for conflicts. Reads may hit the LibDiamond client-side cache or require a wide-area access to the Diamond cloud, while writes (and increments, appends, etc.) are buffered in the cache until commit.

4.3 Transaction Coordination Overview

Figure 5 shows the coordination needed across LibDiamond clients, front-end servers and back-end storage for both read-write and reactive transactions. This section briefly describes the transaction protocols.

Diamond uses *timestamp ordering* to enforce isolation across LibDiamond clients and back-end storage; it assigns every read-write transaction a unique *commit timestamp* that is provided by a replicated *timestamp service* (tss) (not shown in Figure 4). Commit timestamps reflect the transaction commit order, e.g., in strict serializability mode, they reflect a single linearizable ordering of committed, read-write transactions. Both Diamond's client-side cache and back-end storage are *multi-versioned* using these commit timestamps.

4.3.1 Running Distributed Transactions

Read-write and reactive transactions execute similarly; however, as Section 5 relates, reactive transactions can commit locally and often avoid wide-area accesses altogether. We lack the space to cover Diamond's transaction protocol in depth; however, it is similar to Spanner's [17] with two key differences: (1) Diamond uses DOCC for concurrency control rather than a locking mechanism, and (2) Diamond uses commit timestamps from the timestamp service (tss) rather than TrueTime [17].

As shown in Figure 5 (left), transactions progress through two phases, *execution* and *commit*. During the execution phase, LibDiamond runs the application code in the transaction closure passed into `txn_execute`. It runs the code locally on the LibDiamond client node (i.e., not on a storage node like a stored procedure).

The execution phase completes when the application exits the transaction closure or calls `txn_commit` explicitly. Reactive transactions commit locally; for read-write transactions, LibDiamond sends the operation sets to the front-end server, which acts as the *coordinator* for a *two-phase commit* (2PC) protocol, as follows:

1. It sends `Prepare` to all participants (i.e., partitions of the Diamond back-end that hold records in the operation sets), which replicate it via VR.

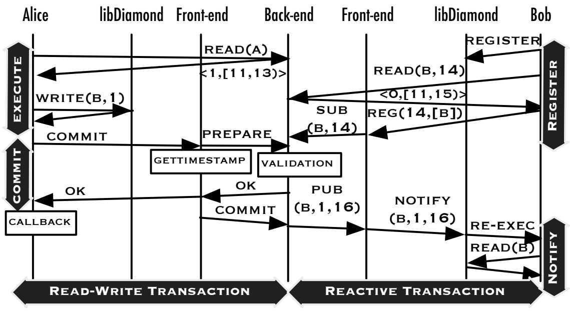


Figure 5: **Diamond transaction coordination.** Left: Alice executes a read-write transaction that reads `A` and writes `B`. Right: Bob registers a reactive transaction that reads `B` (we omit the `txn_id`). When Alice commits her transaction, the back-end server publishes the update to the front-end, which pushes the notification and the update to Bob's LibDiamond, which can then re-execute the reactive transaction locally.

2. Each participant runs a DOCC validation check (described in Section 5); if DOCC validation succeeds, the participant adds the transaction to a *prepared list* and returns `true`; otherwise, it returns `false`.
3. As an optimization, the front-end server concurrently retrieves a commit timestamp from the tss.
4. If all participants respond `true`, the front-end sends `Commit`s to the participants with the commit timestamp; otherwise, it sends `Abort`s. Then, it returns the transaction outcome to the LibDiamond client.

When the client receives the response, it logs the transaction outcome and invokes the transaction callback.

4.3.2 Managing Reactive Transactions

As shown in Figure 5 (right), when an application registers a reactive transaction, the LibDiamond client: (1) gives the reactive transaction a `txn_id`, (2) executes the reactive transaction at its latest known timestamp, and (3) sends the `txn_id`, the timestamp, and the read set in a `Register` request to the front-end server. For each key in the read set, the front-end server creates a `Subscribe` request and sends those requests, along with the timestamp, to each key's back-end partition.

For efficiency, LibDiamond tracks read set changes between executions and re-registers. We expect each reactive transaction's read set to change infrequently, reducing the overhead of registrations; if it changes often, we can use other techniques (e.g., `map_objectrange` described in Section 6.2) to improve performance.

When read-write transactions commit, Diamond executes the following steps for each updated record:

1. The leader in the partition sends a `Publish` request with the transaction's commit timestamp to each front-end subscribed to the updated record.
2. For each `Publish`, the front-end server looks up the reactive transactions that have the updated record in their read sets and checks if the commit timestamp

is bigger than the last notification sent to that client.

3. If so, the front-end server sends a `Notify` request to the client with the commit timestamp and the reactive transaction id.
4. The client logs the notification on receipt, updates its latest known timestamp, and re-executes the reactive transaction at the commit timestamp.

For keys that are updated frequently, back- and front-end servers batch updates. Application clients can bound the batching latency (e.g., to 5 seconds), ensuring that reactive transactions refresh at least once per batching latency when clients are connected.

4.3.3 Handling Failures

While both the back-end storage and tss are replicated using VR, Diamond can suffer failures of the LibDiamond clients or front-end servers. On client failure, LibDiamond runs a *client recovery protocol* using its transaction log to ensure that read-write transactions eventually commit. For each completed but unprocessed transaction (i.e., in the log but with no outcome), LibDiamond retries the commit. If the cloud store has a record of the transaction, it returns the outcome; otherwise, it re-runs 2PC. For each reactive transaction, the application re-registers on recovery. LibDiamond uses its log to find the last timestamp at which it ran the transaction.

Although front-end servers are stateless, LibDiamond clients must set up a new front-end server connection when they fail. They use the client recovery protocol to do this and re-register each reactive transaction with its latest notification timestamp. Front-end servers also act as coordinators for 2PC, so back-end storage servers use the *cooperative termination protocol* [11] if they do not receive `Commit` requests after some timeout.

5 Wide-area Optimizations

This section discusses Diamond's optimizations to reduce wide-area overhead.

5.1 Data-type Optimistic Concurrency Control

Diamond uses an optimistic concurrency control (OCC) mechanism to avoid locking across wide-area clients. Unfortunately, OCC can perform poorly across the wide area due to the higher latency between a transaction's read of a record and its subsequent commit. This raises the likelihood that a concurrent write will invalidate the read, thereby causing a transaction abort. For example, to increment a counter, the transaction reads the current value, increments it, and then commits the updated value; if another transaction attempts the same operation at the same time, an abort occurs.

DOCC tackles this issue in two ways. First, it uses fine-grained concurrency control based on the *semantics* of reactive data types, e.g., allowing concurrent updates to different list elements. Second, it uses conflict-free data

Table 4: **DOCC validation matrix.** Matrix shows whether the committing transaction can commit (C) or must abort (A) on conflicts. Each column is further divided by the isolation level (RC=read committed, SI=snapshot isolation, SS=strict serializability). Commutative CRDT operations have the same outcome.

Prepared Op / **Isolation Level** / **Commiting Op**	**read**			**write**			**CRDT op**		
	RC	**SI**	**SS**	**RC**	**SI**	**SS**	**RC**	**SI**	**SS**
read	C	C	C	C	C	A	C	C	A
write	C	C	A	C	A	A	C	A	A
CRDT op	C	C	A	C	A	A	C	C	C

types with commutative operations, such as counters and ordered sets. As noted in Section 4.3.1, LibDiamond collects an operation set for every data type operation during the transaction's execution phase. For each operation, it collects the key and table. It also collects the read version for every `Get`, the written value for every `Put`, the index (e.g., list index or hash table key) for every collection operation, and the diff (e.g., the increment value or the insert or append element) for every commutative CRDT operation. We show in Section 6 that although fine-grained tracking slightly increases DOCC overhead, it improves overall performance.

Using operation sets, DOCC runs a *validation* procedure that checks every committing transaction for potential violations of isolation guarantees. A *conflicting access* occurs for an operation if the table, key, and index (for collection types) match an operation in a prepared transaction. For a read, a conflict also occurs if the latest write version (or commutative CRDT operation) to the table, key, and index is bigger than the read version. For each, DOCC makes an abort decision, as noted in Table 4.

Since transactions that contain only commutative operations can concurrently commit, DOCC can allow many concurrent transactions that modify the same keys. This property is important for workloads with high write contention, e.g., the Twitter "like" counter for popular celebrities [36]. Further, because Diamond runs read-only and reactive transactions in serializable snapshot mode, they do not conflict with read-write transactions with commutative CRDT operations.

5.2 Client Caching with Bounded Validity Intervals

Some clients in the wide-area setting may occasionally be unavailable, making it impossible to atomically invalidate all cache entries on every write to enforce strong ordering. Diamond therefore uses multi-versioning in both the client-side cache and back-end storage to enforce a *global ordering of transactions*. To do this, it tags each version with a *validity interval* [62], which begins at the *start timestamp* and is terminated by the *end timestamp*. In Diamond's back-end storage, a version's start timestamp is the commit timestamp of the transaction that wrote the

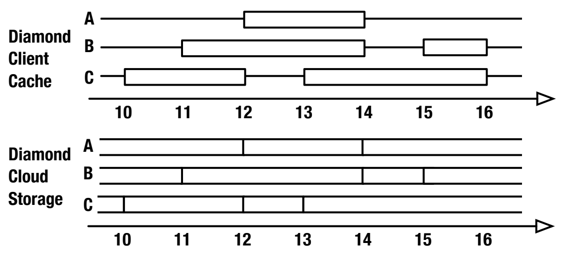


Figure 6: **Diamond versioned cache.** Every Diamond client has a cache of the versions of records stored by the Diamond cloud storage system. The bottom half shows versions for three keys (A, B and C), and the top half shows cached versions of those same keys. Note that the cache is missing some versions, and all of the validity intervals in the cache are bounded.

version. The end timestamp is either the commit timestamp of the transaction writing the *next* version (making that version out-of-date) or *unbounded* for the latest version. Figure 6 shows an example of back-end storage with three keys.

On reads, the Diamond cloud tags the returned value with a validity interval for the LIBDIAMOND client-side cache. These validity intervals are conservative; back-end storage *guarantees* that the returned version is valid *at least* within the validity interval, although it may be valid beyond. If the version is the latest, back-end storage will *bound* the validity interval by setting the end timestamp to the latest commit timestamp of a transaction that accessed that record. For example, in Figure 6, the validity interval of the latest version of B and C are capped at timestamp 16 in the cache, while they are unbounded in storage. Most importantly, bounded validity intervals *eliminate* the need for cache invalidations because the version is always valid within the validity interval. Diamond eventually garbage collects cached versions as they become too outdated to use.

5.3 Data Push Notifications

Reactive transactions require many round-trips to synchronously fetch each update; these can be expensive in a wide-area network. Fortunately, unlike stand-alone notifications services (e.g., Thialfi), Diamond has insight into what data the application is likely to access when the reactive transaction re-executes. Thus, Diamond uses *data push notifications* to batch updates along with notifications, reducing wide-area round trips.

When front-end servers receive `Publish` requests from back-end storage, they perform a snapshot read of every key in the reactive transaction's last read set at the updating transaction's commit timestamp, then piggyback the results with the `Notify` request to the LIBDIAMOND client. LIBDIAMOND re-executes the reactive transaction at the commit timestamp; therefore, if its read set has not changed, then it requires no additional wide-area requests. Further, since the reads were done at the commit timestamp, LIBDIAMOND knows that the transaction can be serialized at that timestamp and committed locally, eliminating all wide-area communication.

Table 5: **Application comparison.** Diamond both reduces code size and adds to the application's ACID+R guarantees.

Application	LoC w/o Diamond	LoC w/ Diamond	LoC Saved	Added A	C	I	D	R
100 Game	46	34	26%	✓	✓	✓		
Chat Room	355	225	33%	✓	✓	✓		✓
PyScrabble	8729	7603	13%	✓			✓	
Twitter clone	14278	12554	13%	✓	✓	✓		

6 Experience and Evaluation

This section evaluates Diamond with respect to both programming ease and performance. Overall, our results demonstrate that Diamond simplifies the design of reactive applications, provides stronger guarantees than existing custom solutions, and supports automated reactivity with low performance overhead.

6.1 Prototype Implementation

We implemented a Diamond prototype in 11,795 lines of C++, including support for C++, Python and Java language bindings on both x86 and ARM. The Java bindings (939 LoC) use javacpp [39], and the Python bindings (115 LoC) use Boost [2]. We cross-compiled Diamond and its dependencies for Android using the NDK standalone toolchain [29]. We implemented most Diamond data types, but not all are supported by DOCC. Our current prototype does not include client-side persistence and relies on in-memory replication for the back-end store; however, we expect disk latency on SSDs to have a low performance impact compared to wide-area network latency, with NVRAM reducing storage latency even further in the future.

6.2 Programming Experience

This section evaluates our experience in building new Diamond apps, porting existing apps to Diamond, and creating libraries to support the needs of reactive programs.

6.2.1 Simplifying Reactive Applications

To evaluate Diamond's programming benefits, we implemented applications both with and without Diamond. Table 5 shows the lines of code for both cases. For all of the apps, Diamond simultaneously decreased program size and added important reliability or correctness properties. We briefly describe the programs and results below.

100 Game. Our non-Diamond version of the 100 game is based on the design in Figure 1. For simplicity, we used Redis [67] for both storage and notifications. We found several data races between storage updates and notifications when running experiments for Figure 9, forcing us to include updates in the notifications to ensure clients did not read stale data from the store. The Diamond version eliminated these bugs and the complexities described in Section 2 and guaranteed correctness with atomicity and isolation; in addition, it reduced the code size by 26%.

Chat Room. As another simple but representative example of a reactive app, we implemented two versions of a chat room. Our version with explicit data management used Redis for storage and the Jetty [40] web server to implement a REST [25] API. It used `POST` requests to send messages and polled using `GET` requests for displaying the log. This design is similar to that used by Twitter [80, 35] to manage its reactive data (e.g., Twitter has `POST` and `GET` requests for tweets, timelines, etc.). The Diamond version used a `StringList` for the chat log, a read-write transaction to append messages, and a reactive transaction to display the log. In comparison, Diamond not only eliminated the need for a server or storage system, it also provided atomicity (the Redis version has no failure guarantees), isolation (the Redis version could not guarantee that all clients saw a consistent view of the chat log), and reactivity (the Redis version polled for new messages). Diamond also shrunk the 355-line app by 130 lines, or 33%.

PyScrabble and Diamond Scrabble. To evaluate the impact of reactive data management in an existing application, we built a Diamond version of PyScrabble [16], an open-source, multiplayer Scrabble game. The original PyScrabble does not implement persistence (i.e., it has no storage system) and uses a centralized server to process moves and notify players. The centralized server enforces isolation and consistency only if there are no failures. We made some changes to add persistence and accommodate Diamond's transaction model. We chose to directly `rmap` the Scrabble board to reactive data types and update the UI in a reactive transaction, so our implementation had to commit and share every update to make it visible to the user; thus, other users could see the player lay down tiles in real-time rather than at the end of the move, as in the original design. Overall, our port of PyScrabble to Diamond removed over 1000 lines of code from the 8700-line app (13%) while transparently simplifying the structure (removing the server), adding fault tolerance (persistence) and atomicity, and retaining strong isolation.

Twimight and Diamond Dove. As another modern reactive application, we implemented a subset of Twitter using an open-source Android Twitter client (Twimight [79]) and a custom back-end. The Diamond version eliminated much of the data management in the Twimight version, i.e., pushing and retrying updates to the server and maintaining consistency between a client-side SQLite [71] cache and back-end storage. Diamond directly plugged into UI elements and published updates with read-write transactions. As a result, it simplified the design, eliminated 1700 lines (13%) from the 14K-line application, transparently provided stronger atomicity and isolation guarantees, and eliminated inconsistent behaviors (e.g., a user seeing a retweet before the original tweet).

6.2.2 Simplifying Reactive Libraries

In addition to simplifying the design and programming of reactive apps, we found that Diamond facilitates the creation of general-purpose reactive libraries. As one example, Diamond transactions naturally lend themselves to managing UI elements. For instance, a check box usually `rmaps` a `Boolean`, re-draws a UI element in a reactive transaction, and writes to the `Boolean` in a read-write transaction when the user checks/unchecks the box. We implemented a general library of Android UI elements, including a text box and check box. Each element required under 50 lines of code yet provided strong ACID+R guarantees. Note that these elements tie the user's UI to shared data, making it impossible to update the UI only locally; for example, if a user wants to preview a message before sharing it with others, the app must update the UI in some other way.

For generality, Diamond makes no assumptions about an app's data model, but we can build libraries using `rmap` for common data models. For example, we implemented object-relational mapping for Java objects whose fields were Diamond data types. Using Java reflection, `rmap_object` maps each Diamond data type inside an object to a key derived from a base key and the field's name. We also support `rmap` for subsets of Diamond collections, e.g., `rmap_range` for Diamond's primitive list types, which binds a subset of the list to an array, and `rmap_objectrange`, which maps a list of objects using `rmap_object`.

These library functions were easy to build (under 75 lines of code) and greatly simplified several applications; for example, our Diamond Twitter implementation stores a user's timeline as a `LongList` of tweet ids and uses `map_objectrange` to directly bind the tail of the user's timeline into a custom Android adapter, which then plugs into the Twimight Android client and automatically manages reactivity. In addition to reducing application complexity, these abstractions also provide valuable hints for prefetching and for how reactive transaction read sets might change. Overall, we found Diamond's programming model to be extremely flexible, powerful, and easy to generalize into widely useful libraries.

6.3 Performance Evaluation

Our performance measurements demonstrate that Diamond's automated data management and strong consistency impose a low performance cost relative to custom-

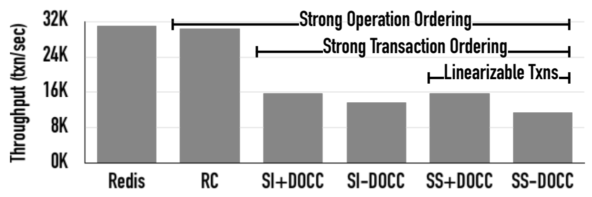


Figure 7: **Peak throughput for explicit data management vs Diamond.** We compare an implementation using Redis and Jetty to Diamond at different isolation levels with and without DOCC. We label the ordering guarantees provided by each configuration. In all cases, the back-end servers were the bottleneck.

written applications. Using transactions with strong isolation properties lowers throughput, as one would expect. We also show that Diamond's DOCC improves performance of transactional guarantees, and that data push notifications reduce the latency of wide-area transactions. Finally, our experiments prove that Diamond has low overhead on mobile devices and can recover quickly from failures.

6.3.1 Experimental Setup

We ran experiments on Google Compute Engine [30] using 16 front-end servers and 5 back-end partitions, each with 3 replicas placed in different availability zones in the same geographic region (US-Central). Our replication protocol used adaptive batching with a batch size of 64. We placed clients in a different geographic region in the same country (US-East). The latency between zones was ≈1 ms, while the latency between regions was ≈36 ms. For our mobile device experiments, we used Google Nexus 7 LRX22G tablets connected via Wi-Fi and, for desktop experiments, we used a Dell workstation with an Intel Xeon E5-1650 CPU and 16 GB RAM.

We used a benchmark based on Retwis [45], a Redis-based Twitter clone previously used to benchmark transactional storage systems [84]. The benchmark was designed to be a representative, although not realistic, reflection of a Twitter-like workload that provides control over contention. It ran a mix of five transactions that range from 4-21 operations, including: loading a user's home timeline (50%), posting a tweet (20%), following a user (5%), creating a new user (1%), and "like"-ing a tweet (24%). To increase contention, we used 100K keys and a Zipf distribution with a co-efficient of 0.8.

6.3.2 Overhead of Automated Data Management

For comparison, we built an implementation of the Retwis benchmark that explicitly manages reactive data using Jetty [40] and Redis [67]. The Redis `WAIT` command offers synchronous in-memory replication, which matches Diamond's fault-tolerance guarantees but provides no operation or transaction ordering [66]. The leftmost bar in Figure 7 shows the peak Retwis throughput of 31K trans./sec. for the Redis-based implementation, while the second bar in Figure 7 shows the Diamond read-committed (RC) version, whose performance (30.5K trans./sec.) is nearly identical. Unlike the Redis-based implementation, however, the Diamond benchmark provides strong consistency based on VR, i.e., it enforces a single global order of operations but not transactions. The Diamond version also provides all of its reactivity support features. Diamond therefore provides better consistency properties and simplifies programming at little additional cost.

As we add stronger isolation through transactions, throughput declines because two-phase commit requires each back-end server to process an extra message per transaction. As the graph shows, snapshot isolation (SI) and strict serializability (SS) reduce throughput by nearly 50% from RC. The graph also shows SI and SS both with and without DOCC; eliminating DOCC hurts SS more than SI (27% vs. 13%) because SI lets transactions with read-write conflicts commit (leading to write skew).

From this experiment, we conclude that Diamond's general-purpose data management imposes almost no throughput overhead. Also, achieving strong transactional isolation guarantees does impose a cost due to the more complex message protocol required. Depending on the application, programmers can choose to offset the cost by allocating more servers or tolerate inconsistencies that result from weaker transactional guarantees.

6.3.3 Benefit of DOCC

DOCC's benefit depends on both contention and transaction duration. To evaluate this effect, we measured the throughput improvement of DOCC for each type of Retwis transaction with at least one CRDT operation (Figure 8).

The `add_user` and `like` transactions are short and thus unlikely to abort, but they still see close to a 2x improvement. `add_follower` gets a larger benefit (4x) because it is a longer transaction with more commutative operations. Even `get_timeline`, a read-only transaction, gets a tiny improvement (2.5%) due to reduced load on the servers from aborting transactions. Further, because `get_timeline` runs in serializable snapshot mode, `post_tweet` transactions can commit concurrently with `get_timeline` transactions.

The `post_tweet` transaction appends a user's new tweet to his timeline and his followers' home timelines (each user has between 5 and 20 followers). If a user follows a large number of people that tweet frequently, conventional OCC makes it highly likely that a conflicting `Append` would cause the entire transaction to fail. With DOCC, all `Appends` to a user's home timeline can commute, avoiding these aborts. As a result, we saw a 5x improvement in abort rate with DOCC over conventional OCC for `post_tweet`, leading to a 25x improvement in throughput. Overall, these results show that Diamond's support for data types in its API and concurrency control mechanism is crucial to reducing the cost of transactional guarantees.

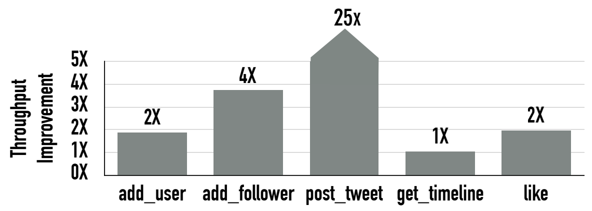


Figure 8: **Throughput improvement with DOCC for each Retwis transaction type.**

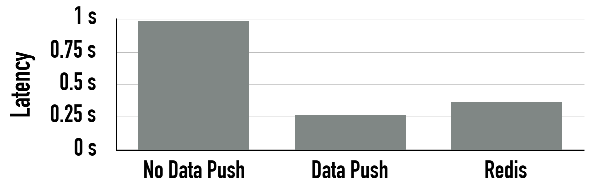


Figure 9: **Latency comparison for 100 game rounds with data push notifications.** Each round consist of 1 move by each of 2 players; latency is measured from 1 client. We implemented explicit data management and notifications using Redis and Diamond notifications with and without batched updates.

6.3.4 Benefit of Data Push Notifications

Although Diamond's automated data management imposes a low throughput overhead, it can hurt latency due to wide-area round trips to the Diamond cloud. For example, the latency of a Retwis transaction is twice as high for Diamond relative to our Redis implementation because Diamond requires two round trips per transaction, one to read and one to commit, while Redis needs only one.

Data push notifications reduce this latency by batching updates with reactive transaction notifications to populate the client-side cache. We turned our implementation of the 100 game from Figure 3 into a benchmark: two players join each game, and players make a move as soon as the other player finishes (i.e., zero "think" time). This experiment is ideal because the read set of the reactive transaction does not change, and it overlaps with the read set of the read-write transaction. We also design an implementation using Redis, where notifications carry updates to clients as a manual version of data push notifications. We measure the latency from one player's client for each player to take a turn or for one *round* of the game. Figure 9 shows that data push notifications reduce the overall latency by almost 50% by eliminating wide-area reads for both the reactive and read-write transactions in the game. As a result, Diamond has 30% lower latency and stronger transactional guarantees than our Redis implementation.

6.3.5 Impact of Wide-area Storage Server Failures

Failures affect the latency of both reactive and read-write transactions. To measure this impact, we used the same 100 game workload and killed a back-end server during the game. To increase the recovery overhead, we geo-replicated the back-end servers across Asia, US-Central and Europe, while clients remained in US-East.

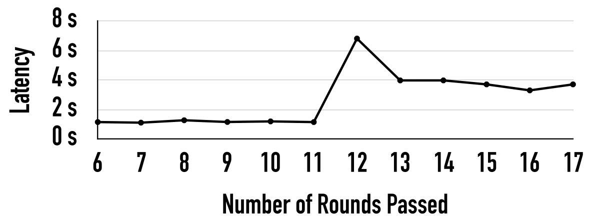


Figure 10: **Latency of 100 game rounds during failure.** We measured the latency for both players to make a move and killed the leader of the storage partition after about 15 seconds. After recovery, the leader moves to another geographic region, increasing overall messaging latency on each move.

Figure 10 shows the latency of each round. Note that the latency is higher than that in the previous experiment because the VR leader has to wait for a response from a quorum of replicas, which take at least 100 ms, and up to 150 ms, to contact. About 15 seconds into the game, we kill the leader in US-Central, switching it to Europe. The latency of each round increases to almost 4 seconds afterwards: the latency between the front-end servers and the leader in Europe increases to 100 ms, and the latency from the leader to the remaining replica in Asia increases to 250 ms. Despite this, the round during the failure takes only 7 seconds, meaning that Diamond can detect the failure and replace the leader in less than 3 seconds.

6.3.6 End-user Application Latency

To evaluate Diamond's impact on the user experience, we measure the latency of user operations in two apps from Section 6.2 built with and without Diamond. PyScrabble is a desktop application, while our Chat Room app runs on Android. The ping times to the Diamond cloud were ≈38 ms on the desktop and ≈46 ms on the Android tablet.

Figure 11 (left) shows two operations for PyScrabble: `MakeMove` commits a transaction that updates the user's move, and `DisplayMove` includes MakeMove plus the notification and reactive transaction to make it visible. Compared to the original PyScrabble, Diamond's latency is slightly higher (9% and 16%, respectively). Figure 11 (right) shows operations for the Chat Room on an Android tablet. `ReadLog` gets the full chat log, and `PostMessage` gets the chat log, appends a message, and commits it back. The Diamond version is a few percent faster than the Redis version because it runs in native C++, while the Redis version uses a Java HTTP client. Overall, we found the latency differences between Diamond and non-Diamond operations were not perceivable to users.

7 Related Work

Diamond takes inspiration from wide-area storage systems, transactional storage systems and databases, reactive programming, distributed programming frameworks, shared memory systems and notification systems.

Several commercial platforms [51, 26, 60] provide

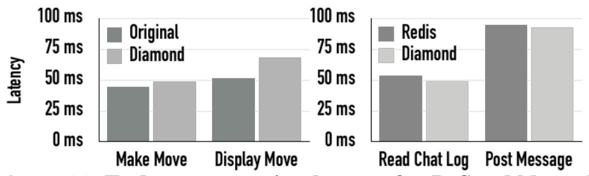


Figure 11: **End-user operation latency for PyScrabble and Chat Room on Diamond and non-Diamond implementations.**

an early form of reactive data management without distributed transactions. Other open source projects [38, 55, 21, 59, 70] have replicated the success of their commercial counterparts. Combined, they comprise a mobile back-end market of $1.32 billion dollars [49].

However, these products do not meet the requirements of reactive applications, still requiring programmers to address failures and race conditions. Meteor [51] lets client-side code directly access the database interface. However, because it uses existing databases (MongoDB [53], and most recently, Postgres [63]) that do not support distributed transactions and offer weak consistency guarantees by default, programmers must still reason about race conditions and consistency bugs. Parse [60] and Firebase [26] similarly enable clients to read, write, and subscribe to objects that are automatically synchronized across mobile devices; however, these systems offer no concurrency control or transactions. As demonstrated by these Stack Overflow questions [56, 50], programmers find this to be a significant issue with these systems. Diamond addresses this clear developer need by providing ACID+R guarantees for reactive applications.

There has been significant work in wide-area storage systems for distributed and mobile applications, including numerous traditional instantiations [77, 42, 57] as well as more recent work [18, 9, 74, 61, 75]. Many mobile applications today use commercial storage services such as Dropbox and others [23, 22, 37], while users can also employ revision-based storage (e.g., git [27]). Applications often combine distributed storage with notifications [3, 6]. As discussed, these systems help with data management, but none offers a complete solution.

Diamond shares a data-type-based storage model with data structure stores [67, 68]. Document stores (e.g., MongoDB [53]) support application objects; this prevents them from leveraging semantics for better performance. These datastores, along with more traditional key-value and relational storage systems [15, 8, 44, 76], were not designed for wide-area use although they could support reactive applications with additional work.

Reactive transactions in Diamond are similar to database triggers [47], events [14], and materialized views [12]. They differ from these mechanisms because they modify local application state and execute application code rather than database queries that update storage state. Diamond's design draws on Thialfi [3]; however, Thialfi cannot efficiently support data push notifications without insight into the application's access patterns.

DOCC is similar to Herlihy [32, 31] and Weihl's [83] work on concurrency control for abstract data types. However, Diamond applies their techniques to CRDTs [69] over a range of isolation levels in the wide area. DOCC is also related to MDCC [43] and Egalitarian Paxos [54]; however, DOCC uses commutativity for transactional concurrency control rather than Paxos ordering and supports more data types. DOCC extends recent work on software transactional objects [33] for single-node databases to the wide area; integrating the two would let programmers implement custom data types in Diamond.

Diamond does not strive to support a fully reactive, data-flow-based programming model, like functional reactive or constraint-based programming [82, 7]; however, reactive transactions are based on the idea of change propagation. Recent interest in reactive programming for web client UIs has resulted in Facebook's popular React.js [64], the ReactiveX projects [65], and Google's Agera[28]. DREAM [48], a recently proposed, distributed reactive platform, lacks transactional guarantees. Sapphire [85], another recent programming platform for mobile/could applications, does not support reactivity, distributed transactions, or general-purpose data management.

8 Conclusion

This paper described Diamond, the first data management service for wide-area reactive applications. Diamond introduced three new concepts: the `rmap` primitive, reactive transactions, and DOCC. Our evaluation demonstrated that: (1) Diamond's programming model greatly simplifies reactive applications, (2) Diamond's strong transactional guarantees eliminate data race bugs, and (3) Diamond's low performance overhead has no impact on the end-user.

9 Acknowledgements

We thank the UW systems lab for their comments throughout the project. This work was supported by Google, National Science Foundation grant CNS-1518702 and NSF GRFP, and MSR Ph.D. fellowships. We also thank our anonymous reviewers and our shepherd, Bryan Ford, for their feedback.

References

[1] Nim, Feb 2016. https://en.wikipedia.org/wiki/Nim#The_100_game.

[2] D. Abrahams and S. Seefeld. Boost C++ libraries, 2015. http://www.boost.org/doc/libs/1_60_0/libs/python/doc/html/index.html.

[3] A. Adya, G. Cooper, D. Myers, and M. Piatek. Thialfi: a client notification service for internet-scale applications. In *Proc. of SOSP*, 2011.

[4] S. Alvos-Bock. The convergence of iOS and OSX user interface design, July 2015. http://www.solstice-mobile.com/blog/the-convergence-of-ios-and-os-x-user-interface-design.

[5] Apple. The Swift programming language, 2016. https://developer.apple.com/library/ios/documentation/Swift/Conceptual/Swift_Programming_Language/#//apple_ref/doc/uid/TP40014097-CH3-ID0.

[6] Apple push notification service, 2015. https://developer.apple.com/library/ios/documentation/NetworkingInternet/Conceptual/RemoteNotificationsPG/Chapters/ApplePushService.html.

[7] K. Apt. *Principles of Constraint Programming*. Cambridge University Press, 2003.

[8] J. Baker, C. Bond, J. C. Corbett, J. Furman, A. Khorlin, J. Larson, J.-M. Leon, Y. Li, A. Lloyd, and V. Yushprakh. Megastore: Providing scalable, highly available storage for interactive services. In *Proc. of CIDR*, 2011.

[9] N. M. Belaramani, J. Zheng, A. Nayate, R. Soulé, M. Dahlin, and R. Grimm. PADS: A policy architecture for distributed storage systems. In *Proc. of NSDI*, 2009.

[10] J. K. Bennett, J. B. Carter, and W. Zwaenepoel. Munin: Distributed shared memory based on type-specific memory coherence. In *Proc. of PPOPP*, 1990.

[11] P. A. Bernstein, V. Hadzilacos, and N. Goodman. *Concurrency Control and Recovery in Database Systems*. Addison Wesley, 1987.

[12] J. A. Blakeley, P.-A. Larson, and F. W. Tompa. Efficiently updating materialized views. In *Proc. of SIGMOD*, 1986.

[13] J. Callaham. Yes, windows 10 is the next version of windows phone. Windows Central, Sept 2014. http://www.windowscentral.com/yes-windows-10-next-version-windows-phone.

[14] S. Chakravarthy. Sentinel: an object-oriented DBMS with event-based rules. In *Proc. of SIGMOD*, 1997.

[15] 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 Transactions on Computer Systems*, 2008.

[16] K. Conaway. Pyscrabble. http://pyscrabble.sourceforge.net/.

[17] J. C. Corbett, J. Dean, M. Epstein, A. Fikes, C. Frost, J. 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 *Proc. of OSDI*, 2012.

[18] M. Dahlin, L. Gao, A. Nayate, A. Venkataramana, P. Yalagandula, and J. Zheng. PRACTI replication. In *Proc. of NSDI*, 2006.

[19] DB-engine's ranking of key-value stores, 10 2015. http://db-engines.com/en/ranking/key-value+store.

[20] 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 *Proc. of SOSP*, 2007.

[21] deepstream. deepstream.io: a scalable server for realtime web apps. https://deepstream.io/.

[22] Google Drive, 2016. http://drive.google.com.

[23] Dropbox, 2015. http://www.dropbox.com.

[24] eMarketer. Mobile game revenues to grow 16.5% in 2015, surpassing $3 billion, Feb 2015. http://www.emarketer.com/Article/Mobile-Game-Revenues-Grow-165-2015-Surpassing-3-Billion/1012063.

[25] R. T. Fielding. *Architectural Styles and the Design of Network-based Software Architectures*. PhD thesis, University of California, Irvine, 2000.

[26] Firebase, 2015. https://www.firebase.com/.

[27] Git, 2015. https://git-scm.com/.

[28] Google. Agera. https://github.com/google/agera.

[29] Google. Android standalone toolchain, 2016. http://developer.android.com/ndk/guides/standalone_toolchain.html.

[30] Google Compute Engine. https://cloud.google.com/products/compute-engine/.

[31] M. Herlihy. Optimistic concurrency control for abstract data types. In *Proc. of PODC*. ACM, 1986.

[32] M. Herlihy. Apologizing versus asking permission: Optimistic concurrency control for abstract data types. *ACM Transactions on Database Systems*, 1990.

[33] N. Herman, J. P. Inala, Y. Huang, L. Tsai, E. Kohler, B. Liskov, and L. Shrira. Type-aware transactions for faster concurrent code. In *eurosys*, 2016.

[34] T. Hoff. Playfish's social gaming architecture - 50 million monthly users and growing. High Scalability, Sept 2010. highscalability.com/blog/2010/9/21/playfishs-social-gaming-architecture-50-million-monthly-user.html.

[35] T. Hoff. The architecture that Twitter uses to deal with 150m active users, 300k qps, a 22 mb/s firehose, and send tweets in under 5 seconds. High Scalability, July 2013. http://highscalability.com/blog/2013/7/8/the-architecture-twitter-uses-to-deal-with-150m-active-users.html.

[36] M. Humphries. Ellen DeGeneres crashes Twitter with Oscar selfie, 2014. http://www.geek.com/mobile/ellen-degeneres-crashes-twitter-with-an-oscars-selfie-1586464/.

[37] Apple iCloud, 2016. https://www.icloud.com/.

[38] A. Incubator. Apache Usergrid. http://usergrid.apache.org/.

[39] JavaCPP: The missing bridge between Java and native C++. github, Mar 2016. https://github.com/bytedeco/javacpp.

[40] Jetty web server. http://www.eclipse.org/jetty/.

[41] R. K. Jonas Boner, Dave Farley and M. Thompson. The reactive manifesto, Sept 2014. http://www.reactivemanifesto.org/.

[42] J. J. Kistler and M. Satyanarayanan. Disconnected operation in the coda file system. *ACM Transactions on Computer Systems*, 1992.

[43] T. Kraska, G. Pang, M. J. Franklin, S. Madden, and A. Fekete. MDCC: multi-data center consistency. In *Proc. of EuroSys*, 2013.

[44] A. Lakshman and P. Malik. Cassandra: A decentralized structured storage system. *ACM SIGOPS Operating Systems Review*, 2010.

[45] C. Leau. Spring Data Redis - Retwis-J, 2013. http://docs.spring.io/spring-data/data-keyvalue/examples/retwisj/current/.

[46] K. Li and P. Hudak. Memory coherence in shared virtual memory systems. *ACM Transactions on Computer Systems*, 1989.

[47] B. F. Lieuwen, N. Gehani, and R. Arlein. The Ode active database: trigger semantics and implementation. In *Proc. of ICDE*, Feb 1996.

[48] A. Margara and G. Salvaneschi. We have a DREAM: Distributed reactive programming with consistency guarantees. In *Proc. of DEBS*. ACM, 2014.

[49] Markets and Markets. Backend as a service (BaaS) market worth 28.10 billion USD by 2020. http://www.marketsandmarkets.com/PressReleases/baas.asp.

[50] martypdx. Firebase data consistency across multiple nodes. Stack Overflow, Apr 2015. http://stackoverflow.com/questions/29947898/firebase-data-consistency-across-multiple-nodes.

[51] Meteor, 2015. http://www.meteor.com.

[52] M. Mode. New mobile apps revolutionize how organizations respond to crises and operations issues, Aug 2014. http://www.missionmode.com/new-mobile-apps-revolutionize-organizations-respond-crises-operations-issues/.

[53] MongoDB, 2015. https://www.mongodb.org.

[54] I. Moraru, D. G. Andersen, and M. Kaminsky. There is more consensus in egalitarian parliaments. In *Proc. of SOSP*, 2013.

[55] Mozilla. Kinto. http://kinto.readthedocs.org/en/latest/.

[56] R. Mulia. Firebase - maintain/guarantee consistency. Stack Overflow, Jan 2016. http://stackoverflow.com/questions/34678083/firebase-maintain-guarantee-data-consistency.

[57] A. Muthitacharoen, B. Chen, and D. Mazières. A low-bandwidth network file system. In *Proc. of SOSP*, 2001.

[58] B. M. Oki and B. H. Liskov. Viewstamped replication: A new primary copy method to support highly-available distributed systems. In *Proc. of PODC*, 1988.

[59] openio. openio.io: object storage grid for apps. http://openio.io/.

[60] Parse, 2015. http://www.parse.com.

[61] D. Perkins, N. Agrawal, A. Aranya, C. Yu, Y. Go, H. V. Madhyastha, and C. Ungureanu. Simba: tunable end-to-end data consistency for mobile apps. In *Proc. of EuroSys*, 2015.

[62] D. R. K. Ports, A. T. Clements, I. Zhang, S. Madden, and B. Liskov. Transactional consistency and automatic management in an application data cache. In *Proc. of OSDI*, 2010.

[63] PostgreSQL, 2013. http://www.postgresql.org/.

[64] React: A JavaScript library for building user interfaces. Github, 2016. https://facebook.github.io/react/.

[65] ReactiveX: An api for asynchronous programming with observable streams, 2016. http://reactivex.io/.

[66] Redis. Wait numslaves timeout. http://redis.io/commands/WAIT.

[67] Redis: Open source data structure server, 2013. http://redis.io/.

[68] Riak, 2015. http://basho.com/products/riak-kv/.

[69] M. Shapiro, N. Preguiça, C. Baquero, and M. Zawirski. Conflict-free replicated data types. In *Proc. of SSS*, 2011.

[70] socketcluster.io. socketcluster.io: a scalable framework for realtime apps and microservices. http://socketcluster.io/#!/.

[71] Sqlite home page, 2015. https://www.sqlite.org/.

[72] Square cash. https://cash.me/.

[73] M. Stonebraker and J. M. Hellerstein. *Readings in Database Systems*. Morgan Kaufmann San Francisco, 1998.

[74] J. Strauss, J. M. Paluska, C. Lesniewski-Laas, B. Ford, R. Morris, and M. F. Kaashoek. Eyo: Device-transparent personal storage. In *Proc. of USENIX ATC*, 2011.

[75] J. Stribling, Y. Sovran, I. Zhang, X. Pretzer, J. Li, M. F. Kaashoek, and R. Morris. Flexible, wide-area storage for distributed systems with WheelFS. In *Proc. of NSDI*, 2009.

[76] R. Sumbaly, J. Kreps, L. Gao, A. Feinberg, C. Soman, and S. Shah. Serving large-scale batch computed data with Project Voldemort. In *Proc. of FAST*, 2012.

[77] D. B. Terry, M. M. Theimer, K. Petersen, A. J. Demers, M. J. Spreitzer, and C. H. Hauser. Managing update conflicts in bayou, a weakly connected replicated storage system. In *Proc. of SOSP*, 1995.

[78] Global social gaming market to reach US$17.4 bn by 2019 propelled by rising popularity of fun games. Transparency Market Research Press Release, Sept 2015. http://www.transparencymarketresearch.com/pressrelease/social-gaming-market.htm.

[79] Twimight open-source Twitter client for Android, 2013. http://code.google.com/p/twimight/.

[80] Twitter. Twitter developer API, 2014. https://dev.twitter.com/overview/api.

[81] Venmo. https://venmo.com/.

[82] Z. Wan and P. Hudak. Functional reactive programming from first principles. In *Proc. of PLDI*, 2000.

[83] W. E. Weihl. Local atomicity properties: modular concurrency control for abstract data types. *ACM Trans. Prog. Lang. Syst.*, 1989.

[84] I. Zhang, N. K. Sharma, A. Szekeres, A. Krishnamurhty, and D. R. K. Ports. Building consistent transactions with inconsistent replication. In *Proc. of SOSP*, 2015.

[85] I. Zhang, A. Szekeres, D. V. Aken, I. Ackerman, S. D. Gribble, A. Krishnamurthy, and H. M. Levy. Customizable and extensible deployment for mobile/cloud applications. In *Proc. of OSDI*, 2014.

Slicer: Auto-Sharding for Datacenter Applications

Atul Adya, Daniel Myers, Jon Howell, Jeremy Elson, Colin Meek, Vishesh Khemani, Stefan Fulger, Pan Gu, Lakshminath Bhuvanagiri, Jason Hunter, Roberto Peon, Larry Kai, Alexander Shraer, Arif Merchant, and Kfir Lev-Ari[†]

Google [†]*Technion - Israel*

Abstract

Sharding is a fundamental building block of large-scale applications, but most have their own custom, ad-hoc implementations. Our goal is to make sharding as easily reusable as a filesystem or lock manager. Slicer is Google's general purpose sharding service. It monitors signals such as load hotspots and server health to dynamically shard work over a set of servers. Its goals are to maintain high availability and reduce load imbalance while minimizing churn from moved work.

In this paper, we describe Slicer's design and implementation. Slicer has the consistency and global optimization of a centralized sharder while approaching the high availability, scalability, and low latency of systems that make local decisions. It achieves this by separating concerns: a reliable data plane forwards requests, and a smart control plane makes load-balancing decisions off the critical path. Slicer's small but powerful API has proven useful and easy to adopt in dozens of Google applications. It is used to allocate resources for web service front-ends, coalesce writes to increase storage bandwidth, and increase the efficiency of a web cache. It currently handles 2-7M req/s of production traffic. The median production Slicer-managed workload uses 63% fewer resources than it would with static sharding.

1 Introduction

Many applications require the resources of more than one computer, especially at Google's typical scale. An application that distributes its work across multiple computers requires some scheme for splitting it up. Often, work is simply split randomly. This is ubiquitous in web services, where the dominant architecture puts a round-robin load-balancer in front of a fleet of interchangeable application processes ("tasks").

However, in many applications, it is hard to ensure that every task can service any request. For example,

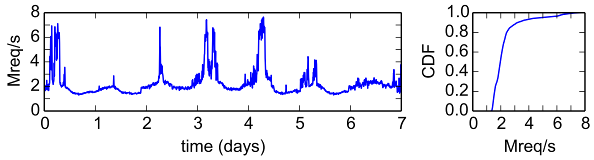


Figure 1: Over five-minute intervals in a recent week, Slicer directed a median of 2 Mreq/s of production traffic with peaks exceeding 7 Mreq/s.

Google's speech recognizer (§3.2.1) uses a different machine learning model for each spoken language. Loading a model is too slow for interactive use: a language must be resident before a request arrives. One task cannot fit every model, making random request balancing untenable. Instead, each task loads only a subset of languages, and incoming requests are routed to a prepared task.

In the past, Google applications like the speech recognizer had their own one-off sharders. Experience taught us that sharding is hard to get right: the plumbing is tedious, and it can take years to tune and cover corner cases. Rebuilding a sharder for every application wastes engineering effort and often produces brittle results.

In practice, custom sharders typically make do with simplistic static sharding that is unresponsive to changes in workload distribution and task availability. Simple schemes utilize resources poorly. In the speech recognizer, resources required per language peak at different times as speakers wake and sleep. When tasks fail, requests must be redistributed among the healthy tasks. When a datacenter fails, a great wave of traffic sloshes over to the remaining datacenters, dramatically altering the request mix. Before Slicer, the speech team handled variation with overprovisioning and manual intervention.

Slicer refactors sharding into a reusable and easily adopted building block akin to a filesystem or lock manager. Slicer is a general-purpose infrastructure service that partitions work across tasks in applications that benefit from affinity. Slicer is minimally invasive to applications: they need only associate incoming requests with a key of their choice that is used to rendezvous requests with tasks. In the speech recognizer, the slice key is the language. Other applications use fine-grained slice keys, such as usernames or URLs. Slicer assigns part of the key space to each task and routes incoming requests to them via integration with Google's front-end load balancers and RPC system.

Slicer addresses these needs by sharding dynamically. It monitors the request load to detect hotspots. It monitors task availability changes due to service provisioning, system updates, and hardware failures. It rebalances the key mapping to maintain availability of all keys and reduce load imbalance among tasks while minimizing key churn.

Slicer can trade off consistency with availability, offering either strongly or eventually consistent assignments. In consistent assignment mode, no task ever believes a key is assigned to it if the Assigner does not agree. The simplest application of this property ensures that at most one task is authoritative for a key, reducing availability but making it easy to write a correct application that mutates state. Alternatively, Slicer can distribute overlapping eventually consistent assignments, eliminating periods of unavailability and reacting rapidly to load shifts.

Slicer's design differs significantly from past sharding systems, driven by its use in dozens of large-scale systems at Google. Slicer provides global optimization and consistency guarantees possible with a centralized load-balancer, but it achieves nearly the same resilience to failures and low latency as systems that make purely local decisions, such as distributed hash tables.

In a production environment, customers cannot tolerate flag days (synchronized restarts). By separating the forwarding data plane from the policy control plane, Slicer simplifies customer-linked libraries and keeps complexity in a central service where the team can more easily coordinate changes.

This functionality is all exposed through a narrow, readily adopted API that has proven useful in Google applications with a variety of needs:

Avoiding storage overhead. A stateless front-end that accesses underlying durable storage on every request is conceptually simple but pays a high performance cost over keeping state in RAM. In some applications, including our speech recognizer, this overhead dwarfs all other time spent serving a user request. For example, a Google pub-sub service[9] processes 600 Kreq/s, most of which do one hash and one comparison to a hash in memory. Fetching the hash via a storage RPC would be correct but incur far more overhead and latency.

Automatic scaling. Many cluster management systems can automatically expand the number of tasks assigned to a job based on load, but these are typically coarse-grained decisions with heavyweight configuration. Our speech recognizer handles dozens of languages, and Slicer's key redundancy provides a single-configuration mechanism to independently scale those many fine-grained resources.

Write aggregation. Several event processors at Google (§3.3.1) ingest huge numbers of small events and summarize them by key (such as data source) into a database. Aggregating writes from stateless front ends is possible, but aggregating like keys on the same task can be more efficient; Data Analysis Pipeline sees 80% fewer storage requests. Affinity provides similar benefits for other expensive, immobile resources like network sockets: Slicer routes requests for an external host to one task with the socket already open.

Sharding state is well-studied; see Section 6. Slicer draws on storage sharding [2, 14, 15] but applies to more classes of application. Compared to other general-purpose sharding systems [5, 10, 8, 13], Slicer offers more features (better load balancing, optional assignment consistency, and key replication) and an architecture focused on high availability.

This paper makes the following contributions:

- An architecture that separates the assignment generation "control plane" from the request forwarding "data plane", which provides algorithmic versatility, high performance, resilience to failure, and exploits existing lease managers and storage systems as robust building blocks.
- An effective load-balancing algorithm that minimizes key churn and has proven effective in a variety of applications.
- An evaluation on production deployments of several large applications that shows the benefits and availability of the Slicer architecture.

2 Slicer Overview and API

Slicer is a general-purpose sharding service that splits an application's work across a set of *tasks* that form a *job* within a datacenter, balancing load across the tasks. A "task" is an application process running on a multi-tenant host machine alongside tasks from other applications. The unit of sharding in Slicer is a key, chosen by the application. Slicer integrates with Google's Stubby RPC system to easily route RPCs originating in other services and with Google's frontend HTTP load balancers to route HTTP requests from external browsers and REST clients.

Slicer has the following components: a centralized *Slicer Service*; the *Clerk*, a library linked into application clients; and the *Slicelet*, a library linked into application server tasks. (Figure 2). The Service is written in Java; the libraries are available in C++, Java, and Go. The Slicer Service generates an *assignment* mapping key ranges ("*slices*") to tasks and distributes it to the Clerks and Slicelets, together called the *subscribers*. The Clerk directs client requests for a key to the assigned task. The Slicelet enables a task to learn when it is assigned or relieved of a slice. The Slicer Service monitors load and task availability to generate new assignments to maintain availability of all keys. Application code interacts only indirectly with the Slicer Service via the Clerk and Slicelet libraries.

2.1 Sharding Model

Application keys may be fine-grained, such as user IDs, or coarse-grained, such as the languages in the speech recognizer described in Section 3.2.1. Keys are an atomic unit of work placement: all state associated with a single key will be collocated on those task replicas to which the key is assigned, but different keys may be assigned to different tasks. Slicer does not observe application state; it merely notifies the task of the keys the task should serve.

Slicer hashes each application key into a 63-bit *slice key*; each slice in an assignment is a range in this hashed keyspace. Manipulating key ranges makes Slicer's workload independent of whether an application has ten keys or a billion and means that an application can create new keys without Slicer on the critical path. As a result, there is no limit on the number of keys nor must they be enumerated.

Hashing keys simplifies the load balancing algorithm because clusters of hot keys in the application's keyspace are likely uniformly distributed in the hashed keyspace.

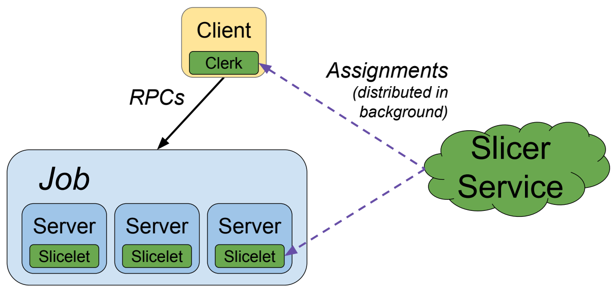


Figure 2: Abstract Slicer architecture.

The cost is lost locality: contiguous application keys are scattered. Many Google applications are already structured around single-key operations rather than scans, encouraged by the behavior of existing storage systems. For others, Section 2.2 offers a mitigation.

Some applications require all requests for the same key to be served by the same task, for example, to maintain a write-through cache. For these, Slicer offers a consistency guarantee on what assignments a Slicelet can observe (§4.5). For many other applications, weaker semantics are correct even when requests for the same key are served by different tasks. For example, such systems serve read-only data (such as Google Fonts), or provide weak consistency to their users (such as Cloud DNS), or have an underlying storage system that provides strong consistency (such as event aggregation systems).

Such applications can configure Slicer with *key redundancy*, allowing assignment of each slice to multiple tasks. Slicer honors a minimum redundancy to protect availability and automatically increases replication for hot slices, which we call *asymmetric key redundancy*.

2.2 Slicelet Interface

The application server task interacts with Slicer via the "Slicelet" API (Figure 3). A simple application, like the Flywheel URL status cache (§3.1.1), is free to ignore this API entirely and answer whatever requests arrive; Slicer transparently improves performance. An application may register a `SliceletListener` to learn when slices arrive and depart, so it can prefetch and garbage-collect state (such as the speech models in Section 3.2.1).

A few affinity-mode applications use `isAffinitizedKey` to discover misrouted requests, such as when retrying a request from the client is cheaper than processing it at the wrong server (§3.3).

```
interface Slicelet {
  boolean isAffinitizedKey(String key);
  Opaque getSliceKeyHandle(String key);
  boolean isAssignedContinuously(Opaque handle);
}
interface SliceletListener {
  void onChangedSlices(List<Slice> assigned,
       List<Slice> unassigned);
}
```

Figure 3: Slicer Server API

To support applications that require exclusive key ownership to maintain consistent in-memory state, the Slicelet provides an API inspired by Centrifuge [10]. The task calls `getSliceKeyHandle` when a request arrives, and passes the handle back to `isAssignedContinuously` before externalizing the result. Note that checking assignment at beginning and end is insufficient, since the slice may have been unassigned and reassigned in the meantime. A task may also cache a handle across multiple requests, for example to cache a user's inbox during a session.

To scan its store to preload state, an application may need to map from hashed slices keys back to original application keys. Applications with few keys (such as language names in the speech recognizer) can precompute an index at each task. Applications with many keys typically adjust their storage schema, either by prefixing the primary key with the hashed slice key or by adding a secondary index. In future work, Slicer will support unhashed application-defined keys and implement range sharding to preserve locality among adjacent application-defined keys.

By default, Slicer load balances on request rate (req/s). The Slicelet integrates with Stubby to transparently monitor request rate per slice. Some applications have highly variable cost per request, or want to balance a different metric like task CPU utilization. An extension to the API of Figure 3 lets tasks report a custom load metric.

2.3 Clerk Interface

The Clerk provides a single function which maps a key to the addresses of its assigned tasks (Figure 4). Most applications ignore this API and simply enable transparent integration with Google's RPC system Stubby or Google's HTTP proxy GFE (Google Front End).

```
interface Clerk {
  Set<Addr> getAssignedTasks(String key);
}
```

Figure 4: Slicer Client API

Stubby typically directs RPCs round-robin from each client to a subset of tasks in a job. We extended Stubby to accept an additional *slice key* argument with each RPC, causing the task to be selected using Slicer's assignment. Stubby also has support for Google's global load balancer, which selects the network-closest datacenter for each RPC. With both enabled, the global load balancer picks a datacenter, and Slicer picks the task from the job in that datacenter.

The GFE is an HTTP proxy that accepts requests from the Internet and routes each to an internal task. The GFE offers a declarative language for selecting routing features from a request's URL, parameters, cookies, headers and more. Slicer integration interprets any such feature as a slice key.

3 Slicer Uses in Production Systems

Slicer is used by more than 20 client services at Google, and it balances 2-7M requests per second with more than 100,000 application client processes and server tasks connected to it (Figure 1). Prospective customers evaluate their systems against a test instance of Slicer that routes another 2 Mreq/s.

This section illustrates some of Slicer's use cases. Current uses of Slicer fit three categories: in-memory cache, in-memory store, and aggregation.

3.1 In-memory Cache Applications

Slicer is most commonly used for in-memory dynamic caches over storage state.

3.1.1 Flywheel

Flywheel is an optimizing HTTP proxy for mobile devices [11]. Flywheel tracks which websites have recently been unreachable, enabling an immediate response to a client that averts a timeout. Flywheel uses a set of "tracker" tasks as a repository of website reachability. In the original design, updates and requests were sent to a random tracker task. Because the semantics are forgiving, this worked but converged slowly. To hasten unreachability detection, Flywheel now uses Slicer with website server name as the key, so that updates and requests converge on a single task.

3.1.2 Other cache uses

Many other services use Slicer to manage caches.

1. *Meeting scheduler*: manages meetings and provides calendar functions. Includes a per-user cache for faster responses.
2. *Crawl manager*: crawls pages and extracts metadata. Retains last crawl time per URL to provide crawl rate-limiting.
3. *Fonts service*: serves fonts to various web and mobile applications. Caches font files and subsets of font files.
4. *Configuration sync service*: periodically checks end-to-end configurations for entities from multiple sources. Entity affinitization allows comparisons of configurations from multiple sources.
5. *Data analysis pipeline*: analyzes stored data and serves summary results. Caches query results per source.
6. *Job profiling*: caches metadata used for job profiling by job name.
7. *User Contacts Cache*: caches user's contacts information when fetched by a user's mobile or web application.
8. *User Metadata Cache*: caches user's metadata/preferences for a user in a video display application.
9. *Service Control*: caches aggregated metrics and logs for public APIs.

3.2 In-memory Store Applications

The in-memory caches in the previous section handle shard reassignment by discarding state, causing future requests to the moved keys to see a cache miss. In contrast, the tasks of an in-memory store load any missing data from an underlying store, and thus resharding events only affect latency; the stored data remains available.

3.2.1 Speech Recognition

As mentioned in Section 1, a speech recognition system uses Slicer to assign languages to tasks and route incoming requests to a task with the required model loaded. The speech team originally manually partitioned languages into task-sized sets and put each set in a separate job. This approach required peak provisioning, failing to multiplex resources to exploit diurnal shifts as populations wake and sleep. It was also operationally complex, incurring manual overhead to monitor, maintain, upgrade, and debug separately-configured jobs.

3.2.2 Cloud DNS

Google's Cloud DNS service, which hosts millions of domains owned by Google and its customers, uses Slicer to assign DNS records to tasks, allowing the tasks to quickly make purely local decisions using in-memory state. Furthermore, Slicer's key redundancy and load balancing support allows the service to respond to load changes in the key space. Since the application provides DNS semantics, Slicer's affinity mode is sufficient.

3.3 Aggregation Applications

Tasks receive requests for some key (e.g., customer id, pubsub topic) and they aggregate them into larger writes to a backing store. This reduces traffic on the underlying store: Event Pipeline 2 achieved a $\frac{4}{5}$ reduction. Slicer's asymmetric key replication is particularly effective for aggregation, spreading hot key traffic across many tasks. The tasks write concurrently and depend on key-granularity append semantics at the store to preserve correctness [14].

3.3.1 Event analysis

Two event analysis systems shard events by source id to build up a model. Without Slicer, these systems would have to read, modify and write the model on every event, since aggregating writes would incur frequent expensive optimistic concurrency control conflicts.

With Slicer, requests for a source id key are almost[1] always routed to the same task. Therefore, a task can afford to aggregate writes coarsely, since write conflicts are rare. It can also cache the last model state it wrote, skipping the read step of read-modify-write unless the backend store detects a conflict. In these systems, traffic per source varies by several orders of magnitude, making load balancing essential.

[1]These services use Slicer's affinity mode, which provides high availability at the cost of perfect consistency (§4.5), relying on the backend store's conflict detection for data consistency.

3.3.2 Client Push: Pubsub System for Mobile Devices

Client Push [3] is a pubsub system that allows mobile clients to subscribe to topics and receive all messages published on that topic. Tasks are sharded by topic; they write subscriptions to a table in which the slice key is the prefix of the storage key. Slicer affinitization improves efficiency by aggregating requests for a range of keys to the storage servers. Slicer's asymmetric replication spreads hot topics across many tasks, avoiding bottlenecks.

4 Slicer Service Implementation

Slicer aims to combine the high-quality, strongly consistent sharding decisions of a centralized system with the scalability, low latency, and fault tolerance associated with local decisions. This section describes how Slicer achieves the best of both worlds.

The Assigner is the core of Slicer's backend service. It collects health, task provisioning, and load signals. It uses its central view of those signals to produce a coherent assignment of work to tasks (§4.4) that is strongly consistent for applications that need it (§4.5).

Though the Slicer Service is conceptually centralized (Figure 2), the implementation is highly distributed (Figure 5). By combining client-side caching, *Distributors*, and *Backup Distributors* that provide a backstop against catastrophic failures, the backend service also achieves scalability (§4.2) and fault tolerance (§4.3) similar to a purely local service.

4.1 Assignment Generation

The Assigner generates assignments using a sharding algorithm described in Section 4.4. To enhance availability, we run the Assigner service in several Google datacenters around the world. Any Assigner may generate an assignment for any job in any datacenter.

Deploying multiple Assigners increases availability but admits the possibility of disagreement. Section 4.5 explains how subscribers can observe consistent assignments. But even for eventually consistent applications, the Assigners should converge, not thrash among competing decisions. To facilitate convergence, Assigners write decisions into optimistically-consistent storage. An Assigner reads the stored assignment, generates a new assignment, and assigns it a monotonic generation number. It writes the new assignment back to storage transactionally conditioned on overwriting the previously read value. If a concurrent write has occurred, the transac-

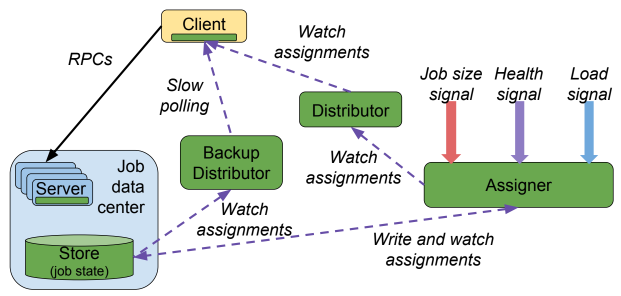


Figure 5: Slicer backend service architecture. The Assigner collects signals and uses them to make an assignment, informed by a stored prior assignment to minimize churn. The Assigner disseminates assignments to subscribers (Clerks and Slicelets) through the distributor and through a passive backup path via a store. All of this traffic is off the critical path of client-server communications. The Assigner and Distributors are replicated across datacenters; each component can serve any job at Google.

tion fails, the Assigner abandons its new assignment, retrieves the new current assignment, and tries again.

For efficiency, in the steady state only a single *preferred* Assigner generates an assignment for a particular job. Each Assigner periodically polls Google's global load balancer service to see if it is network-closest, and hence preferred, for the jobs for which it is generating assignments. This definition is eventually consistent: there may be brief periods when multiple Assigners are preferred.

Assignment storage makes the distributed Assigners act as a single logical process. When failure causes a change in preferred Assigner, the new one learns the decisions of the prior one and carries them forward. Should two Assigners both believe they are preferred, they will thrash, but storage concurrency control prevents divergence.

Slicer makes assignments for one job in one datacenter at a time. Customers who run jobs in multiple datacenters use a higher-level Google load balancer to route a request to a datacenter, and then within that datacenter, use Slicer to pick one task from the job.

4.2 Scalable Assignment Distribution

Because Slicer manipulates ranges of a hashed keyspace, assignments have a concise representation. Even then, large applications with thousands of tasks produce large assignments that need to be distributed to all server tasks and their clients (together, the *subscribers*). This distribution must occur quickly after assignment change. At large scales, distribution becomes a computational and network bottleneck. We address it with a two-tier distribution tree: an Assigner generates and distributes an assignment to a tier of *Distributors*, which distribute it to the subscribers. Nothing in our model precludes adding an additional tier to the tree.

Distribution is a pull model: a subscriber asks a Distributor for a job's assignment; if the Distributor doesn't have it, the distrubutor asks the Assigner, which generates and distributes the assignment. Each Clerk and Slicelet library maintains a long-lived stream with the Distributor service using Google's standard load balancer service, which routes its stream to the closest available instance.

Assignment distribution is asynchronous. Affinity applications can tolerate temporary inconsistency, and consistent applications ensure consistency via a separate control channel (§4.5).

This architecture admits running Distributors in datacenters close to subscribers to minimize WAN traffic. In practice, to ease administration, we currently tolerate the WAN traffic and run Distributors in the same datacenters as Assigners.

Evolving Slicer is easier if we decouple our release schedule from those of our customers. One design alternative we rejected was to have Slicer's subscriber library coordinate peer-to-peer assignment distribution among customer tasks. The cost is that the Slicer team must provision its own resources for assignment distribution, but the benefit is to minimize logic linked into customer binaries. Likewise, putting the logic that identifies the preferred Assigner in the Distributor tier keeps it out of subscriber libraries.

4.3 Fault Tolerance

We've designed Slicer to maintain request routing despite failures of infrastructure and of Slicer itself. Slicer's control-plane separation ensures that most failures merely hinder timely re-optimization of the assignment, yet requests continue to flow. The rest of this section enumerates properties of the system which achieve these goals.

Backup Assignment Retrieval Path. When an application client or server task starts, it must fetch the current assignment through the network of Distributors. The Distributors share a nontrivial code base and thus risk a correlated failure due to a code or configuration error. We have yet to experience such a correlated failure, but our paranoia and institutional wisdom motivated us to guard against it.

Hence the Slicer Service includes a *Backup Distributor* which satisfies application requests simply by reading the assignment from the store (§4.1). The Backup

Distributor is simple, slowly evolving, and mostly independent of the Distributor and Assigner code base.

If the Backup Distributor is the only one operating, the system degrades to static sharding based on slightly stale load and health information. This mode requires only:

1. Library code linked into application binaries,
2. the Backup Distributor service, and
3. a valid assignment in persistent storage.

Because it does not react to load shifts or server task failure, degraded mode is intended as a stopgap until an on-call engineer restores the Assigner and Distributor network.

Geographic Diversity. Distributors and Assigners run in datacenters around the world. Any subscriber can reach any Distributor via the Google global load balancing service, and likewise any Distributor can reach any Assigner. If the preferred Assigner for a job has failed, any Assigner can become preferred. This diversity tolerates machine, datacenter, and network failures.

Geographic Proximity. The preferred Assigner for each job is the Assigner network-closest to the job (§4.1), and a Distributor runs wherever there is an Assigner; these decisions reduce dependence on WAN connectivity. If customers demanded it, Slicer Service could run in every customer cell, eliminating all cross-datacenter dependency.

Fate-Shared Storage Placement. Although no production customers are configured this way, Slicer's implementation allows storing assignments in the same datacenter as the job. By also placing an Assigner in the same datacenter, the job can tolerate a network partition of the datacenter.

Service-Independent Mode. Ultimately, even if every component of the Slicer Service fails, requests continue to flow using the most recent assignment cached in applicaion libraries. This mode has the same limitations as the Backup Distributor mode, plus new or restarted application client tasks are unable to initialize.

In summary, Slicer's design tolerates machine, datacenter, and network failures including complete datacenter partitions. It degrades gracefully under correlated bug and configuration faults that destroy the Assigners, Distributors, or the entire Slicer Service.

4.4 Load Balancing

The ultimate goal of load balancing is to minimize peak load; this enables a service to be provisioned with fewer resources. We balance load because we do not know the future: unexpected surges of traffic arrive at abritary tasks. Maintaining the system in a balanced state maximizes the buffer between current load and capacity for each task, buying the system time to observe and react.

Slicer's initial assignment divides the keyspace equally among available tasks, assuming that key load is uniform (key distribution is uniform due to hashing). If there is variation in either the rate at which different keys receive requests or in the resources required to satisfy those requests, some tasks may become overloaded while others are underutilized. Slicer monitors key load – either request rate, which can be automatically tracked via the Slicelet integration with Stubby, or application-reported custom metrics – to determine if load balancing changes are required. The primary goal of load balancing is to minimize the *load imbalance*, which we define as the ratio of the maximum task load to the mean task load. In a perfectly balanced job where each task is handling the same load, the imbalance is 1.

To provide intuition for the definition: the worst case imbalance Slicer can cause is n/r, where r is the job's minimum key redundancy configuration and n is the task count. For example, with $n = 10$ and $r = 2$, the worst decision Slicer can make is to direct Stubby to route every key to one of two tasks, giving a load imbalance value of 5.

Load imbalance can be reduced by adding or removing redundant tasks for a key or by reassigning keys from one task to another. Besides reducing imbalance, Slicer must respect configurations constraining the minimum and maximum number of tasks that may be assigned to a key. It should also limit *key churn*, the fraction of the key space affected by reassignment. Key churn itself creates load and increases overhead.

To scale to billions of keys, Slicer represents assignments compactly with key ranges. Hence sometimes it must *split* a hot slice—replace a key range $[a,c)$ with two ranges $[a,b),[b,c)$—so that its load can be distributed among multiple tasks. To prevent unbounded assignment size growth, Slicer must also create opportunities to *merge* slices. It does so by assigning adjacent cool slices to the same tasks, then merging the slice representations into a single range.

At Google, independent mechanisms (sometimes humans) decide when to add or remove tasks from a job, or add or remove CPU or memory from tasks in a job. Thus Slicer focuses exclusively on redistributing imbalanced load among available tasks, not on reprovisioning resources for sustained load changes.

4.4.1 Sharding Algorithm: Weighted-move

When Slicer determines that resharding is necessary, due to changing load metrics or changes to the set of tasks in the job, it produces a new assignment using the sharding algorithm, which proceeds in the following phases:

1. Reassign keys away from tasks that are no longer part of the job (e.g., due to hardware failure).
2. Increase/decrease key redundancy as required to conform to configured constraints (e.g. due to a change in the configuration).
3. *Merge* adjacent cold slices, moving one onto the same task as the other, to defragment the assignment. This step proceeds as long as
   (a) there are more than 50 slices per task in aggregate,
   (b) merging two slices creates a slice with less than mean slice load,
   (c) merging two slices does not drive the receiving task's load above the maximum task load, and
   (d) no more than 1% of the keyspace has moved.
4. In this phase, the sharding algorithm picks a sequence of *moves* with the highest *weight*, which we define as the reduction in load imbalance for the tasks affected by the move (benefit) divided by the *key churn* (cost). Moves are applied to the assignment in descending weight order until a key churn budget (9% of the keyspace) is exhausted.
5. *Split* hot slices without changing their task assignments. Splitting captures finer-grained load measurements and opens new move options in the next round. This step proceeds as long as
   (a) the split slice is at least twice as hot as the mean slice, and
   (b) there are fewer than 150 slices per task in aggregate.

In each iteration of phase 4, only moves affecting the hottest task can reduce load imbalance (as defined above), and for each slice in the hottest task, three possible moves are considered: reassigning the slice to the coldest task to displace the load, redundantly assigning the slice to the coldest task to spread the load, or removing the slice which offsets the load to existing assignees. Note that increasing or decreasing assignment redundancy may be illegal given the configuration for the job, so some moves are disqualified. The algorithm greedily makes the best move and repeats until the key churn (cost) budget is exhausted. Successive iterations of the loop may affect different tasks as prior moves revise the estimate of which task is "hottest".

The constants in the algorithm (50–150 slices per task, 1% and 9% key movement per adjustment) were chosen by observing existing applications. Experience suggests the system is not very sensitive to these values, but we have not measured sensitivity rigorously. Future work will estimate application-specific churn cost to better tune the cost-benefit tradeoff.

4.4.2 Rebalancing suppression

Slicer balances request rate, task CPU utilization, or an application-specified custom metric. When balancing CPU and the maximum task load is less than 25% (an arbitrary threshold), Slicer suppresses rebalancing: Because no task is at risk of overload, churn is waste.

4.4.3 Limitations

When balancing the request rate, Slicer ignores task heterogeneity: one task may be cool with 10,000 req/s but another is swamped. CPU utilization balancing inherently adjusts for such heterogeneity.

Some applications make high memory demands for each key. If Slicer colocates many infrequently requested keys on one task, that task may exhaust memory despite manageable CPU load. Our future work will include measuring memory usage and honoring constraints in the algorithm.

4.4.4 A rejected design alternative

A variant of consistent hashing [22] with load balancing support [10] yielded both unsatisfactory load balancing and large, fragmented assignments. We refer to this scheme as *load-aware consistent hashing*. Some applications had too few slice keys (tens to hundreds per task) for consistent hashing to result in good statistical load balancing.

Consistent hashing enables very compact assignments, so long as the client carries the decoding algorithm. Since evolving clients is burdensome (§4.2), Slicer instead distributes assignments in decoded form. Consistent hashing works best with many (1000) virtual nodes per physical task but introduces a significant cost distributing decoded assignments.

More importantly, consistent hashing gives us less control over hot spots. We can cool off a task by reducing its virtual node count, but the displaced traffic ends up randomly distributed, not directed at a cool task, giving a poor tradeoff between key movement and balance improvement.

We were originally drawn to the statelessness of consistent hashing: it produces the same output from the same inputs, which allowed recovering from an Assigner failure without requiring access to the previous assignment. In practice, once the Assigner begins balancing load, creating a profitable reassignment requires knowledge of the previous assignment, and thus it is important that a recovering Assigner have access to prior state.

The load-aware consistent hashing algorithm we abandoned is similar to that in Centrifuge [10]. It was more sophisticated in that it supported key replication, asymmetric replication, and proportional response to imbalance for faster reaction. After 18 months in service, we replaced it with the weighted-move algorithm, which balances better with less key churn (§5.2.1).

4.5 Strong Consistency

An application that needs to maintain data consistency can do so by building upon Slicer's optional *assignment consistency*. It defines an authoritative assignment for every moment and guarantees that no task ever believes a key is assigned to it if that assignment does not agree. By configuring the job for at most one replica of each key, at no time will two Slicelets believe they are both assigned the same key. The consistency feature is implemented, but it is not yet deployed by customers in production.

The simplest way to provide strong consistency guarantees for keys would be to allocate a lease for each key from a central lease manager. We opted against this model, because it would require provisioning lease manager resources in proportion to the number of keys, and hundreds of millions of keys per sharded job are common. Existing lease managers such as Chubby [12] do not scale to that level, so this would require building a highly available, scalable lease manager and running it in every datacenter at Google, which is a non-trivial effort.

While insufficiently scalable to provide a lease per key, Chubby is highly available (the code is battle-tested, and the system has its own operations team) and present in every data center at Google. Slicer builds on Chubby to provide a scalable lease-per-key abstraction using only three Chubby locks per job. The scheme ensures that only the keys being reassigned are unavailable during an assignment change. The design preserves the robustness of Slicer's data plane, so that even if Slicer Service is down, RPCs continue to flow with strong consistency, since lease granting and maintenance is performed by the highly-available and battle-tested Chubby. The Assigner is only required for resharding.

The following describes how the leases provide strong consistency.

To protect the work done while changing a strongly-consistent assignment, an Assigner acquires the exclusive *job lease* to ensure that exactly one Assigner performs the work for writing. If the Assigner crashes during the assignment-change operation, another Assigner can acquire the job lease and resume the unfinished work. Only Assigners interact with the job lease.

To achieve consistent assignment, the Assigner distributes assignments in the usual way, then writes the assignment generation number as the value of the *guard lease*. A consistent Slicelet may only use an assignment once it acquires the guard lease for reading. Clerks require no lease, since the only harm of a transient inconsistent assignment at the Clerk is a misrouted request bounced back for retry.

Changing the assignment entails recalling the guard lease from Slicelet readers so the Assigner can rewrite its value. In any large-scale system, recalling a lease often means waiting out the expiration period for any task that may have died while holding its lease. This recall period entails complete application unavailability.

We make the observation that when an assignment A_1 is replaced by A_2, there is no reason to make unavailable the unchanged slices, those that have identical assignments in $A_1 \cap A_2$. A third *bridge* lease bridges over the transition from A_1 to A_2, making $A_1 \cap A_2$ available during the gap. The Assigner writes and distributes assignment A_2, creates the bridge lease, delays for Slicelets to acquire the bridge lease for reading, and only then does it recall and rewrite the guard lease. A Slicelet is allowed to use the intersection if it holds the bridge lease.

For a synthetic benchmark, we measured a median lease recall period of 2.6 s and 99th percentile period of 4.1 s, implying that absent a bridge lease an entire application would suffer seconds of unavailability whenever an assignment changes. Section 5.2.5 reports on a benchmark that demonstrates how the bridge lease improves availability.

Nothing about the consistent-assignment mechanism limits it to the simple consistency propery of at most one Slicelet per key; the Assigner could easily enforce an at-most-three policy. The simpler policy is easy for applications to exploit, whereas allowing plural replicas would require the application to consistently coordinate those replicas, perhaps with state machine replication [24].

5 Evaluation

This section evaluates Slicer using both measurements from the deployed system and experiments with real and synthetic workloads.

5.1 Production Measurements

We measure production customers to evaluate Slicer's availability, load balancing, scale, and assignment convergence time.

5.1.1 Availability

As the primary – but pessimistic – measure of production availability, we evaluated the integration of Slicer and Stubby. Specifically, we considered how often Slicer was able to select a task for a Stubby client issuing an RPC. Normally, Stubby selects any task in the destination job which the client locally believes to be healthy. With Slicer, Stubby selects a healthy task from the set of Slicer-provided candidates. If all tasks are unhealthy or no assignment is available, the selection fails.

Over a one-week period, Slicer performed 260 billion task selections for a subset of its Stubby clients, of which 99.98% succeeded. This value underestimates the availability of Slicer, because some of the failures may have been because all tasks were unhealthy, and ordinary Stubby would also have failed to select a task, but we expect that such cases are rare. Thus, in those cases where standard Stubby could have sent an RPC, then Stubby with Slicer could have sent an RPC at least 99.98% of the time.

We also examine availability at the server side. In another week, we observed 272 billion requests arrive at server tasks, of which only 11.6 million (0.004%) had been misrouted. This measure overestimates availability because it only considers requests that made it to a server task, and it underestimates availability because many applications can tolerate misdirected requests with only an impact on latency or overhead, not availability.

A secondary measure of availability is that of the Slicer service itself. Our production monitoring periodically requests an assignment from each Distributor instance. In one week, 99.75% of 329,978 requests succeeded. This probe underestimates availability because it requires computation of a new assignment, whereas the common path returns a cached one.

These measurements are over an admittedly short window, limited by production monitoring data retention policy. That said, they indicate Slicer is a suitable building block for highly available applications.

5.1.2 Load balancing

We evaluate how well Slicer balances load across tasks, how much key movement it incurs, and how much it improves over static strategies.

Figure 6 shows the effectiveness of load balancing for several production customer jobs belonging to three services. Sampling five minute windows over a six hour period, we measure the number of requests each task handles, normalized as a fraction of the mean request count for all tasks in the job during the window. The vast majority of time windows had values close to the mean, indicating that the tasks were well-balanced. Peak loads varied between 1.3× – 2.8× the mean load.

Figure 7 shows key churn for tasks in the same jobs as in Figure 6. Churn counts the number of key-moves: one key moving ten times in one hour produces the same value as ten keys moving once. Here we see a broader range of values, as some jobs exhibit higher variance over time (e.g., Cloud DNS, which moves up to 40% of its keys per hour), and some are quite stable over time (e.g., Flywheel, which moves only 16% of its keys). We report fraction of keyspace but not bytes of objects actually unloaded and reloaded because, by design, Slicer does not know which keys in the key space actually exist, nor is it aware of the data associated with those keys (§2.1).

Our production monitoring captured a shift from load-aware consistent hashing to the weighted move algorithm. Figure 8 shows the request rate per task for the general-purpose key-value cache discussed in Section 3.1 during the rollout of the weighted-move algorithm. Under consistent hashing, the hottest task was 50% hotter than the mean. The weighted move algorithm improves the balance, enabling operations engineers to make tighter capacity planning decisions.

Ultimately, customers care about Slicer's load balancing because it offers a big win over home-brew alternatives. We observed production key distributions and load distributions for all customer jobs. We built a model to infer the load on the tasks had the load been balanced

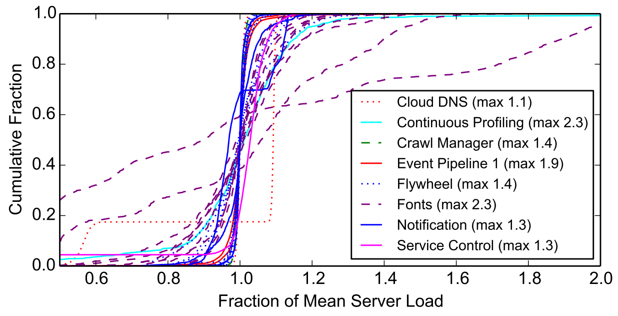


Figure 6: Slicer successfully balances load: tasks in a job rarely experience load 5% greater than the mean task load.

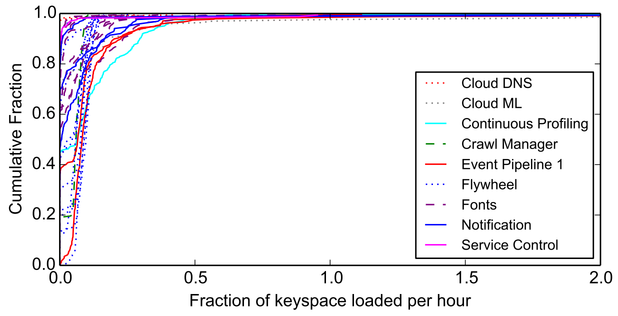


Figure 7: Key movement costs for jobs belonging to customer production services, sampled over one week. The median hour in every job sees less than 20% of the keyspace move.

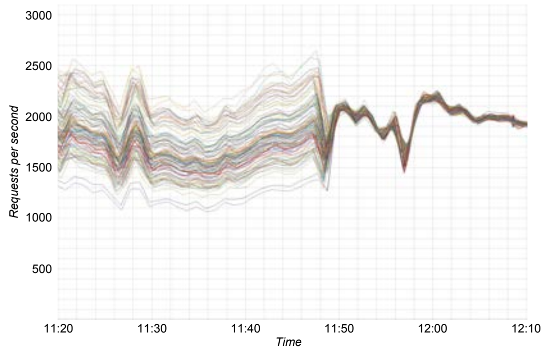


Figure 8: Load per task on a production key/value cache when switching from load-aware consistent hashing to the weighted-move algorithm at 11:50.

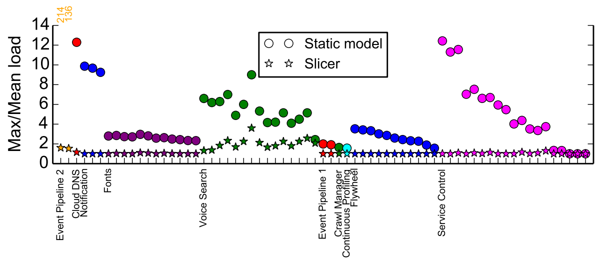


Figure 9: Load balance for production jobs grouped by service, contrasted with a static model. Slicer makes the median job's hottest task 63% less loaded.

statically. If the customer supplied an initial load estimate, the model uses it; otherwise it spreads the keyspace uniformly across tasks. The model mitigates random clumping by partitioning the keyspace into 100 slices per task.

Figure 9 contrasts, for each job, the actual load imbalance under Slicer versus the load imbalance under the static model. Load imbalance is the ratio between the CPU load of the most loaded task and the mean CPU load across tasks. Each pair of points shows the most imbalanced hour in a one-week observation. For underloaded jobs, Slicer defers load balancing, and thus acts identically to the static model; Figure 9 elides such jobs. Service operators provision for peak loads; Slicer provides a median reduction of 63% and as much as 99.3% for the most skewed job.

5.1.3 Scale

Slicer serves more than 20 unique systems (§3). Each is a unique software stack that integrates Slicer in a different way. This table extracts aggregate statistics from production monitoring.

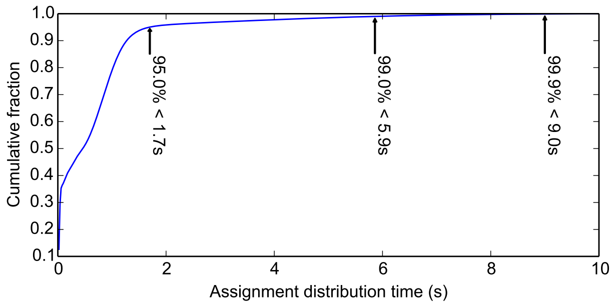


Figure 10: Once emitted by the Assigner, 95% of assignments reach subscribers within 2 s.

Services	22	mean /	mean /
Jobs	263	service	job
Tasks (Slicelets)	11387	517	43
Clerks	113,338	5151	430
Requests/sec	6M	266K	22K
Assignments/hour	662	30	2.5
Assignment traffic (MBps)	180	8.2	0.37
Key churn/hour	4%		

Presently the production Slicer Service includes six Assigners provisioned with three cores each. Sampling one minute windows on each task over one week, the median sample utilizes 0.13 core, and the 99th percentile utilizes 2.34 cores. Considering the entire Service—Assigners, Distributors, Backup Distributors—Slicer uses 0.3% of the CPU and 0.2% of the RAM used by the sliced services and their clients.

5.1.4 Assignment Convergence Time

It is desirable for Slicer to effect assignment changes rapidly, to minimize the period of divergence among subscribers. Figure 10 shows the CDF of assignment distribution latencies across affinity-mode production customers for one week. Assignments generally arrive within the second.

5.1.5 Assignment Computation Time

Most production assignments take a fraction of a second to compute; the 64th percentile is 17ms and the maximum a few seconds.

5.2 Experiments

Experiments in this section explore details and trade-offs under controlled conditions.

5.2.1 Comparing load balancing strategies

We recorded slice keys for RPCs issued to three production users of Slicer: Client Push (see Section 3.3.2), Cloud DNS (see Section 3.2.2) and Flywheel (see Section 3.1.1). We then replayed these requests against three algorithms: static uniform sharding (in which the

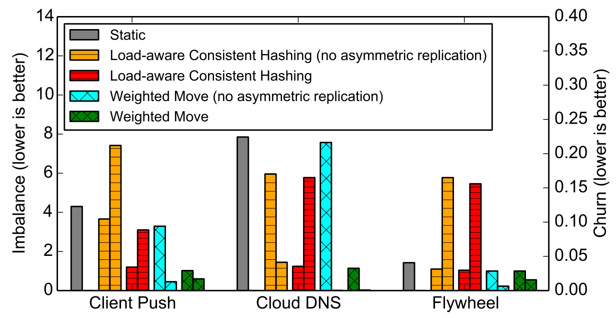


Figure 11: Slicer's centralized weighted move algorithm balances better than static and load-aware consistent hashing schemes, and churns less than load-aware consistent hashing.

key space is divided uniformly amongst all tasks), load-aware consistent hashing (see Section 4.4.4) and Slicer's weighted-move algorithm (see Section 4.4.1). In addition, we compared the performance of the algorithms with and without asymmetric key redundancy (not applicable for static sharding which cannot dynamically assign keys to additional tasks).

Figure 11 shows the mean across all resharding decisions for measurements of *load imbalance*, the ratio of the max task load to the mean task load, and of *key churn*, the fraction of the key space reassigned (both defined in §4.4). Slicer's algorithm – weighted-move with redundancy – significantly outperforms both other algorithms on load imbalance, with reduced key churn relative to consistent hashing (though not static sharding, which being static has no key churn). Asymmetric replication provides significant load balancing benefits, though with a small increase in key churn (due to increased opportunities to address imbalance).

Note that this experiment isolates the impact of load balancing from other factors such as task failures and pre-emption.

5.2.2 Assigner Failure and Recovery

To evaluate Slicer's robustness to Assigner failure, we presented power-law skewed load to twenty tasks. Once the system stabilized, we killed the Assigner task, causing clients and server tasks to continue using the last-generated assignment. After 2 hours, we restored the Assigner.

Results are shown in Figure 12. The pre-failure and post-recovery curves are essentially identical: the Assigner rebalanced load upon recovery. The outage curve shows degraded load balancing, since the assignment stagnated while the load changed. However, the recent static balance is better than uniform sharding (not shown in Figure 12) on the same workload. Production workloads tend to be more stable over time; the outage curve

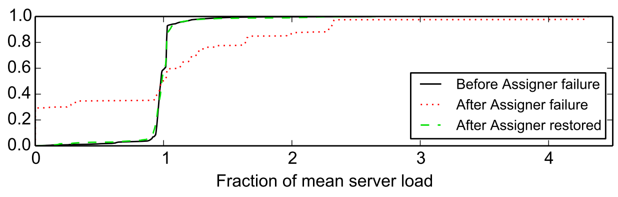


Figure 12: Load balancing before, during, and after an Assigner failure.

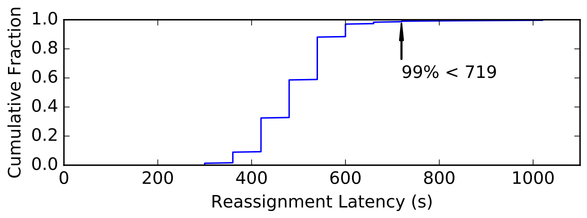


Figure 13: The Assigner typically effects a response to a load shift in 480 seconds.

for such workloads should remain closer to the actively-balanced curves.

In practice, if an Assigner fails, any other Assigner can pick up the slack. We configured a test job with two Assigners, killed the active one, and observed that the other became initialized 17.1 s later ($\sigma = 2.7\ s$). This delay is the period of polls to the Google load balancer for preferred Assigner checks (§4.1).

5.2.3 Load Reaction Time

How quickly does Slicer respond to a load shift? In this experiment, five client tasks offer 8 Kreq/s of synthetic load to ten server tasks, consisting of 100 keys in a power-law distribution with exponent 1.5. Every nineteen minutes, the clients' distribution shifts to move the hottest load to different keys. We report the latency from clients shifting load to tasks reporting a max/mean load imbalance below 1.2. Figure 13 shows a median delay of 480 s, which is a function of the 1 m delay from the Google monitoring system and Slicer's 5 m load observation window. One window is insufficient because, unless the load shifts very early in the window, Slicer's first observation doesn't convince it to shift enough load to completely restore balance.

5.2.4 Scaling Benchmark

One of Slicer's essential architectural decisions is central decisionmaking and a distributed data plane. In the experiment in Figure 14, we contrast Slicer's plumbing with a natural alternative that indirects routing decisions to a centralized *authority*. In the centralized version, clients preface each request with a request to the authority, and server tasks contact the authority on each request to confirm the routing decision. Here the authority is implemented as a single Clerk task relaying

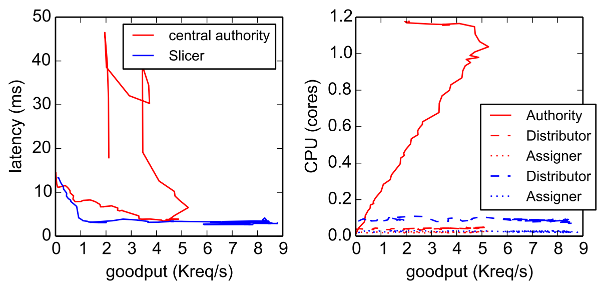


Figure 14: On the left is a latency-throughput curve, on the right CPU consumption versus retired load. Once the central authority saturates its CPU allocaton at 5 Kreq/s, it encounters a scaling limit. This simple experiment lacks admission control, so the throughput drops under overload; a production system would hit the same wall more gracefully.

decisions from an Assigner and Distributor, although a real centralized system would simply colocate load balancing with the authority interface. In both cases, a set of 2000 clients simulated on 100 tasks offers increasing load against 50 server tasks. The authority saturates its CPU at 5 Kreq/s, but Slicer scales smoothly since every component's workload is independent of aggregate client request rate.

5.2.5 Consistency Benchmark

Section 4.5 described how Slicer preserves availability in consistent assignment mode by using bridge leases to carry unchanged key assignments across the distribution period of a new assignment. We evaluate its importance under a synthetic dynamically skewed workload in which 25 clients drive 43 Kreq/s against 50 server tasks. Over three days, 99.85% of requests were satisfied; absent bridging, only 99.19% of requests would have been satisfied.

The `getSliceKeyHandle` operation takes 153 μs and `isAssignedContinuously` takes 94 μs.

6 Related Work

As a general purpose sharding system, Slicer is similar to Centrifuge [10], Orleans [13], Ringpop [8], and Microsoft Service Fabric [5].

It is most similar to Centrifuge, which also uses a central manager, assigns ranges of a hashed keyspace, and provides leases. Slicer differs in four respects. First, Slicer's architecture is more available. If the Centrifuge manager is unavailable, all leases expire and no RPCs flow. Slicer's control-plane separation ensures that assignments remain valid and RPCs flow even if the entire Slicer service fails. Slicer's separate backup distribution path keeps working even when the service is down. Second, Slicer's separation of assignment distiribution from assignment generation enables much higher scales: an Assigner can servie 10^4 Distributors, and a Distributor 10^4 subscribers. Third, Centrifuge is a single-cluster system. Slicer's Assigner can be accessed from a different cluster, enabling failover across clusters. Fourth, Slicer's load balancing is better than Centrifuge's. Slicer moved from Centrifuge-style consistent hashing (S4.4.4) to the weighted-move algorithm (§4.4). It achieves better balance while moving an order-of-magnitude fewer keys (§5), works for both many and few application keys, and creates more compact assignments. Slicer's load balancing supports custom metrics and key redundancy.

As compared to Orleans and Ringpop, Slicer uses a centralized algorithm rather than a client-based consistent hashing [22], which allows it to provide better load balancing and to offer consistency guarantees, which those systems cannot. Service Fabric does not support dynamic sharding: sharding must be specified by the application and cannot be adjusted on the fly to balance load [6]. Additionally, both Orleans and Service Fabric are frameworks and are more invasive to applications than Slicer's small API.

As a sharding manager, Slicer also has elements in common with sharding managers embedded in storage systems. For example, Bigtable [14], HBase [2], and Spanner [15] are all structured in terms of ranges of an application-defined keyspace. Only the HBase algorithm is publically described; it has several strategies, all of which have splits and moves as base operations. Unlike Slicer, it does not support key redundancy or balancing on application-defined metrics. Moreover, storage system sharding managers are not usable outside of the storage system, and they often make storage-specific assumptions that limit their flexibility. For example, Bigtable requires at most one task per tablet to enforce consistency, whereas Slicer is free to add redundant copies of keys if permitted by the application

Social Hash [28] makes cluster-level sharding decisions for HTTP requests and storage systems. Slicer shares Social Hash's separation of coordinated central decisionmaking from distributed forwarding. Where Slicer treats keys independently, Social Hash optimizes placement using inter-key locality available in social graphs. Slicer operates at fine granularity in space (tasks) and in time (seconds to minutes). Slicer supports a wide variety of applications and supports consistent assignment.

BASIL [19] and Kunkle [23] balance I/O workloads in large-scale storage systems; like Slicer, they perform what-if planning and evaluate migrating hot data. They differ from Slicer in several important respects. First, they place a relatively small number of items, and they have application-specific load data for each item. For example, BASIL places virtual disks within a storage array. This is a different problem than in Slicer, which places a potentially vast number of items (e.g., hundreds of millions), is agnostic to the application, and can only collect information at coarse granularity. Second, Slicer has a larger space of possible load balancing moves available; in addition to migrating slices, it can also split and merge them, and it can add or remove redundant copies. Besides minimizing imbalance, Slicer's algorithm also minimizes assignment fragmentation.

In theory, sufficiently fast storage available to all front-ends can sometimes obviate the need to cache sharded data in the front-end. Caches such as Memcached [4] and Redis [7] as well as in-memory stores such as RamCloud [25] and Dynamo [16] can be used. However, remote storage always adds the cost of (un)marshalling data along with a network roundtrip to access data. In addition, such solutions do not help when the shared resource isn't state, such as a network socket. Finally, eliminating external caches and collocating data with code reduces how many services must be provisioned and maintained.

Sharding solutions have been recently proposed [29, 27, 20] for specific databases, focusing on dynamic load balancing (as opposed to balancing the number of keys per task). Accordion [27] places partitions but does not modify their boundaries and thus cannot handle hot data. SPORE [20] replicates hot keys but does not support dynamic task membership or key migration. EStore [29] is a dynamic sharding manager that like SPORE identifies hot keys and migrates them, but it does does not support key redundancy. When hot keys cool down, EStore migrates previously hot keys back to their original shards, which creates unnecessary churn.

Software and hardware network load balancers [17, 26, 18, 21, 1] employ one or more controllers that either process messages themselves or program a set of distributed switches to carry out a load balancing policy. Such load balancers may have a notion of affinity or session "stickiness". However, such balancers implement static hashing for requests or sessions; when they react to load shifts, they do not maximize affinity. The do not provide server tasks with early assignment signals to facilitate prefetching, or termination signals to facilitate garbage collection. They do not offer asymmetric key redundancy, nor do they enable assignment consistency.

7 Conclusions

Slicer is a highly available, low-latency, scalable and adaptive sharding service that remains decoupled from customer binaries and offers optional assignment consistency. These features and the consequent architecture were driven by the needs of real applications at Google. Slicer makes it easy to exploit sharding affinity and has proven to offer a diversity of benefits, such as object caching, write aggregation, and socket aggregation, to dozens of deployed applications.

Production deployment of Slicer shows that the system meets its load balancing and availability goals. Real applications experience a max:mean load ratio of 1.3–2.8, assisting peak load capacity planning. Slicer balances load better than load-aware consistent hashing, and does so while creating an order of magnitude less key churn. Slicer is available, correctly routing production customer requests at least 99.98% of the time, making it a building block for highly-available applications. Adoption by over 20 projects with a variety of use cases demonstrates the generality of its API.

References

[1] Amazon ELB. `https://aws.amazon.com/elasticloadbalancing/`.

[2] Apache HBase. `https://hbase.apache.org/`.

[3] Firebase topic messaging. `https://firebase.google.com/docs/cloud-messaging/android/topic-messaging`.

[4] Memcached. `https://memcached.org/`.

[5] Microsoft service fabric. `https://azure.microsoft.com/en-us/documentation/services/service-fabric/`.

[6] Partitioning in microsoft service fabric. `https://azure.microsoft.com/en-us/documentation/articles/service-fabric-concepts-partitioning/`.

[7] Redis. `http://redis.io/`.

[8] Uber ringpop. `https://eng.uber.com/intro-to-ringpop/`.

[9] 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 (SOSP)*, pages 129–142, 2011.

[10] A. Adya, J. Dunagan, and A. Wolman. Centrifuge: Integrated lease management and partitioning for cloud services. In *Proceedings of the 7th USENIX conference on Networked systems design and implementation*, pages 1–1. USENIX Association, 2010.

[11] V. Agababov, M. Buettner, V. Chudnovsky, M. Cogan, B. Greenstein, S. McDaniel, M. Piatek, C. Scott, M. Welsh, and B. Yin. Flywheel: Google's data compression proxy for the mobile web. In *12th USENIX Symposium on Networked Systems Design and Implementation (NSDI 15)*, pages 367–380, 2015.

[12] M. Burrows. The Chubby lock service for loosely-coupled distributed systems. In *Proc. of OSDI*, 2006.

[13] S. Bykov, A. Geller, G. Kliot, J. R. Larus, R. Pandya, and J. Thelin. Orleans: cloud computing for everyone. In *ACM SOCC*, 2011.

[14] 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 Transactions on Computer Systems (TOCS)*, 26(2):4, 2008.

[15] J. Corbett et al. Spanner: Google's globally distributed database. *ACM Trans. Comput. Syst.*, 31(3), Aug. 2013.

[16] G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, and W. Vogels. Dynamo: amazon's highly available key-value store. In *ACM SIGOPS Operating Systems Review*, volume 41, pages 205–220. ACM, 2007.

[17] D. E. Eisenbud, C. Yi, C. Contavalli, C. Smith, R. Kononov, E. Mann-Hielscher, A. Cilingiroglu, B. Cheyney, W. Shang, and J. D. Hosein. Maglev: A fast and reliable software network load balancer. In *13th USENIX Symposium on Networked Systems Design and Implementation (NSDI 16)*, pages 523–535, 2016.

[18] 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. *ACM SIGCOMM Computer Communication Review*, 44(4):27–38, 2015.

[19] A. Gulati, C. Kumar, I. Ahmad, and K. Kumar. BASIL: Automated io load balancing across storage devices. In *File and Storage Technologies (FAST)*, 2010.

[20] Y.-J. Hong and M. Thottethodi. Understanding and mitigating the impact of load imbalance in the memory caching tier. In *Proceedings of the 4th Annual Symposium on Cloud Computing*, SOCC '13, pages 13:1–13:17, New York, NY, USA, 2013. ACM.

[21] N. Kang, M. Ghobadi, J. Reumann, A. Shraer, and J. Rexford. Efficient traffic splitting on commodity switches. In *Conference on Emerging Networking Experiments and Technologies (CoNEXT)*, 2015.

[22] D. R. Karger, E. Lehman, F. T. Leighton, R. Panigrahy, M. S. Levine, and D. Lewin. Consistent hashing and random trees: Distributed caching protocols for relieving hot spots on the world wide web. In *STOC*, pages 654–663, 1997.

[23] D. Kunkle and J. Schindler. A load balancing framework for clustered storage systems. In *High Performance Computing-HiPC 2008*, pages 57–72. Springer, 2008.

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

[25] 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.

[26] P. Patel, D. Bansal, L. Yuan, A. Murthy, A. Greenberg, D. A. Maltz, R. Kern, H. Kumar, M. Zikos, H. Wu, et al. Ananta: cloud scale load balancing. *ACM SIGCOMM Computer Communication Review*, 43(4):207–218, 2013.

[27] M. Serafini, E. Mansour, A. Aboulnaga, K. Salem, T. Rafiq, and U. F. Minhas. Accordion: Elastic scalability for database systems supporting distributed transactions. *Proceedings of the VLDB Endowment*, 7(12):1035–1046, 2014.

[28] A. Shalita, B. Karrer, I. Kabiljo, A. Sharma, A. Presta, A. Adcock, H. Kllapi, and M. Stumm. Social Hash: An assignment framework for optimizing distributed systems operations on social networks. In *13th USENIX Symposium on Networked Systems Design and Implementation (NSDI 16)*, pages 455–468, Santa Clara, CA, Mar. 2016. USENIX Association.

[29] R. Taft, E. Mansour, M. Serafini, J. Duggan, A. J. Elmore, A. Aboulnaga, A. Pavlo, and M. Stonebraker. E-store: Fine-grained elastic partitioning for distributed transaction processing systems. *Proceedings of the VLDB Endowment*, 8(3):245–256, 2014.

History-Based Harvesting of Spare Cycles and Storage in Large-Scale Datacenters

Yunqi Zhang[†*] George Prekas[‡*] Giovanni Matteo Fumarola[δ]
Marcus Fontoura[δ] Íñigo Goiri[⋆] Ricardo Bianchini[⋆]

[†]*University of Michigan* [‡]*EPFL* [δ]*Microsoft* [⋆]*Microsoft Research*

Abstract

An effective way to increase utilization and reduce costs in datacenters is to co-locate their latency-critical services and batch workloads. In this paper, we describe systems that harvest spare compute cycles and storage space for co-location purposes. The main challenge is minimizing the performance impact on the services, while accounting for their utilization and management patterns. To overcome this challenge, we propose techniques for giving the services priority over the resources, and leveraging historical information about them. Based on this information, we schedule related batch tasks on servers that exhibit *similar* patterns and will likely have enough available resources for the tasks' durations, and place data replicas at servers that exhibit *diverse* patterns. We characterize the dynamics of how services are utilized and managed in ten large-scale production datacenters. Using real experiments and simulations, we show that our techniques eliminate data loss and unavailability in many scenarios, while protecting the co-located services and improving batch job execution time.

1 Introduction

Motivation. Purchasing servers dominates the total cost of ownership (TCO) of large-scale datacenters [4], such as those operated by Google and Microsoft. Unfortunately, the servers' average utilization is often low, especially in clusters that host user-facing, interactive services [4, 10]. The reasons for this include: these services are often latency-critical (*i.e.*, require low tail response times); may exhibit high peaks in user load; and must reserve capacity for unexpected load spikes and failures.

An effective approach for extracting more value from the servers is the co-location of useful batch workloads (*e.g.*, data analytics, machine learning) and the data they require on the same servers that perform other functions, including those that run latency-critical services. However, for co-location with these services to be acceptable, we must shield them from any non-trivial performance interference produced by the batch workloads or their storage accesses, even when unexpected events occur. If co-location starts to degrade response times, the scheduler must throttle or even kill (and re-start elsewhere) the culprit batch workloads. In either case, the performance of the batch workloads suffers. Nevertheless, co-location ultimately reduces TCO [37], as the batch workloads are not latency-critical and share the same infrastructure as the services, instead of needing their own.

Recent scheduling research has considered how to carefully select which batch workload to co-locate with each service to minimize the potential for interference (most commonly, last-level cache interference), *e.g.* [9, 10, 25, 42]. However, these works either assume simple sequential batch applications or overlook the resource utilization dynamics of real services. Scheduling data-intensive workloads comprising many distributed tasks (*e.g.*, data analytics jobs) is challenging, as scheduling decisions must be made in tandem for collections of these tasks for best performance. The resource utilization dynamics make matters worse. For example, a long-running workload may have some of its tasks throttled or killed when the load on a co-located service increases.

Moreover, no prior study has explored in detail the co-location of services with data for batch workloads. Real services often leave large amounts of spare storage space (and bandwidth) that can be used to store the data needed by the batch workloads. However, co-locating storage raises even more challenges, as the management and utilization of the services may affect data durability and availability. For example, service engineers and the management system itself may reimage (reformat) disks, deleting all of their data. Reimaging typically results from persistent state management, service deployment, robustness testing, or disk failure. Co-location and reimaging may cause all replicas of a data block to be

[*]This work was done while Zhang and Prekas were interns at MSR.

destroyed before they can be re-generated.

Our work. In this paper, we propose techniques for harvesting the spare compute cycles and storage space in datacenters for distributed batch workloads. We refer to the original workloads of each server as its "primary tenant", and to any resource-harvesting workload (*i.e.*, batch compute tasks or their storage accesses) on the server as a "secondary tenant". We give priority over each server's resources to its primary tenant; secondary tenants may be killed (in case of tasks) or denied (in case of storage accesses) when the primary tenant needs the resources.

To reduce the number of task killings and improve data availability and durability, *we propose task scheduling and data placement techniques that rely on historical resource utilization and disk reimaging patterns.* We logically group primary tenants that exhibit similar patterns in these dimensions. Using the utilization groups, our scheduling technique schedules related batch tasks on servers that have *similar* patterns and enough resources for the tasks' expected durations, and thereby avoids creating stragglers due to a lack of resources. Using the utilization and reimaging groups, our data placement technique places data replicas in servers with *diverse* patterns, and thereby increases durability and availability despite the harvested nature of the storage resources.

To create the groups, we characterize the primary tenants' utilization and reimaging patterns in ten production datacenters,[1] including a popular search engine and its supporting services. Each datacenter hosts up to tens of thousands of servers. Our characterization shows that the common wisdom that datacenter workloads are periodic is inaccurate, since often most servers do not execute interactive services. We target *all* servers for harvesting.

Implementation and results. We implement our techniques into the YARN scheduler, Tez job manager, and HDFS file system [11, 29, 36] from the Apache Hadoop stack. (Primary tenants use their own scheduling and file systems.) Stock YARN and HDFS assume there are no external workloads, so we also make these systems aware of primary tenants and their resource usage.

We evaluate our systems using 102 servers in a production datacenter, with utilization and reimaging behaviors scaled down from it. We also use simulations to study our systems for longer periods and for larger clusters. The results show that our systems (1) can improve the average batch job execution time by up to 90%; and (2) can reduce data loss by more than two orders of magnitude when blocks are replicated three times, eliminate data loss under four-way replication, and eliminate data unavailability for most utilization levels.

Finally, we recently deployed our file system in large-scale production (our scheduler is next), so we discuss our experience and lessons that may be useful to others.

Summary and conclusions. Our contributions are:

- We characterize the dynamics of how servers are used and managed in ten production datacenters.
- We propose techniques for improving task scheduling and data placement based on the historical behavior of primary tenants and how they are managed.
- We extend the Hadoop stack to harvest the spare cycles and storage in datacenters using our techniques.
- We evaluate our systems using real experiments and simulations, and show large improvements in batch job performance, data durability, and data availability.
- We discuss our experience with large-scale production deployments of our techniques.

We conclude that resource harvesting benefits significantly from a detailed accounting of the resource usage and management patterns of the primary workloads. This accounting enables higher utilization and lower TCO.

2 Related Work

Datacenter characterization. Prior works from datacenter operators have studied selected production clusters, not entire datacenters, *e.g.* [37]. Instead, we characterize *all* primary tenants in ten datacenters, including those used for production latency-critical and non-critical services, for service development and testing, and those awaiting use or being prepared for decommission.

Harvesting of resources without co-location. Prior works have proposed to harvest resources for batch workloads in the absence of co-located latency-critical services, *e.g.* [22, 23]. Our work focuses on the more challenging co-location scenario in modern datacenters.

Co-location of latency-critical and batch tasks. Recent research has targeted two aspects of co-location: (1) performance isolation – ensuring that batch tasks do not interfere with services, after they have been co-located on the same server [19, 20, 21, 24, 27, 31, 32, 38, 42]; or (2) scheduling – selecting which tasks to co-locate with each service to minimize interference or improve packing quality [9, 10, 12, 25, 37, 43]. Borg addresses both aspects in Google's datacenters, using Linux cgroup-based containers, special treatment for latency-critical tasks, and resource harvesting from containers [37].

Our work differs substantially from these efforts. As isolation and interference-aware scheduling have been well-studied, we leave the implementation of these techniques for future work. Instead, we reserve compute resources that cannot be given to batch tasks; a spiking primary tenant can immediately consume this reserve until our software can react (within a few seconds at most) to replenish the reserve. Combining our systems with finer grained isolation techniques will enable smaller reserves.

[1]For confidentiality, we omit certain information, such as absolute numbers of servers and actual utilizations, focusing instead on coarse behavior patterns and full-range utilization exploration.

Moreover, unlike services at Google, our primary tenants "own" their servers, and do not declare their potential resource needs. This means that we must harvest resources carefully to prevent interference with latency-critical services and degraded batch job performance. Thus, we go beyond prior works by understanding and exploiting the primary tenants' resource usage dynamics to reduce the need for killing batch tasks. With respect to resource usage dynamics, a related paper is [5], which derives Service-Level Objectives (SLOs) for resource availability from historical utilization data. We leverage similar data but for dynamic task scheduling, which their paper did not address.

Also importantly, we are the first to explore in detail the harvesting of storage space from primary tenants for data-intensive batch jobs. This scenario involves understanding how primary tenants are managed, as well as their resource usage.

For both compute and storage harvesting, we leverage primary and secondary tenants' historical behaviors, which are often more accurate than user annotations/estimates (e.g., [35]). Any system that harvests resources from latency-critical workloads can benefit from leveraging the same behaviors.

Data-processing frameworks and co-location. Researchers have proposed improvements to the Hadoop stack in the absence of co-location, *e.g.* [3, 8, 13, 14, 15, 18, 39]. Others considered Hadoop (version 1) in co-location scenarios using virtual machines, but ran HDFS on dedicated servers [7, 30, 41]. Lin *et al.* [22] stored data on dedicated and volunteered computers (idle desktops), but in the absence of primary tenants. We are not aware of studies of Mesos [16] in co-location scenarios. Bistro [12] relies on static resource reservations for services, and schedules batch jobs on the leftover resources. In contrast to these works, we propose dynamic scheduling and data placement techniques for the Hadoop stack, and explore the performance, data availability, and data durability of co-located primary and secondary tenants.

3 Characterizing Behavior Patterns

We now characterize the primary tenants in ten production datacenters. In later sections, we use the characterization for our co-location techniques and results.

3.1 Data sources and terminology

We leverage data collected by AutoPilot [17], the primary tenant management and deployment system used in the datacenters. Under AutoPilot, each server is part of an *environment* (a collection of servers that are logically related, *e.g.* indexing servers of a search engine) and executes a *machine function* (a specific functionality, *e.g.* result ranking). Environments can be used for production, development, or testing. In our terminology, each primary tenant is equivalent to an <environment, machine function> pair. Primary tenants run on physical hardware, *without* virtualization. Each datacenter has between a few hundred to a few thousand primary tenants.

Though our study focuses on AutoPilot-managed datacenters, our characterization and techniques should be easily applicable to other management systems as well. In fact, similar telemetry is commonly collected in other production datacenters, *e.g.* GWP [28] at Google and Scuba [2] at Facebook.

3.2 Resource utilization

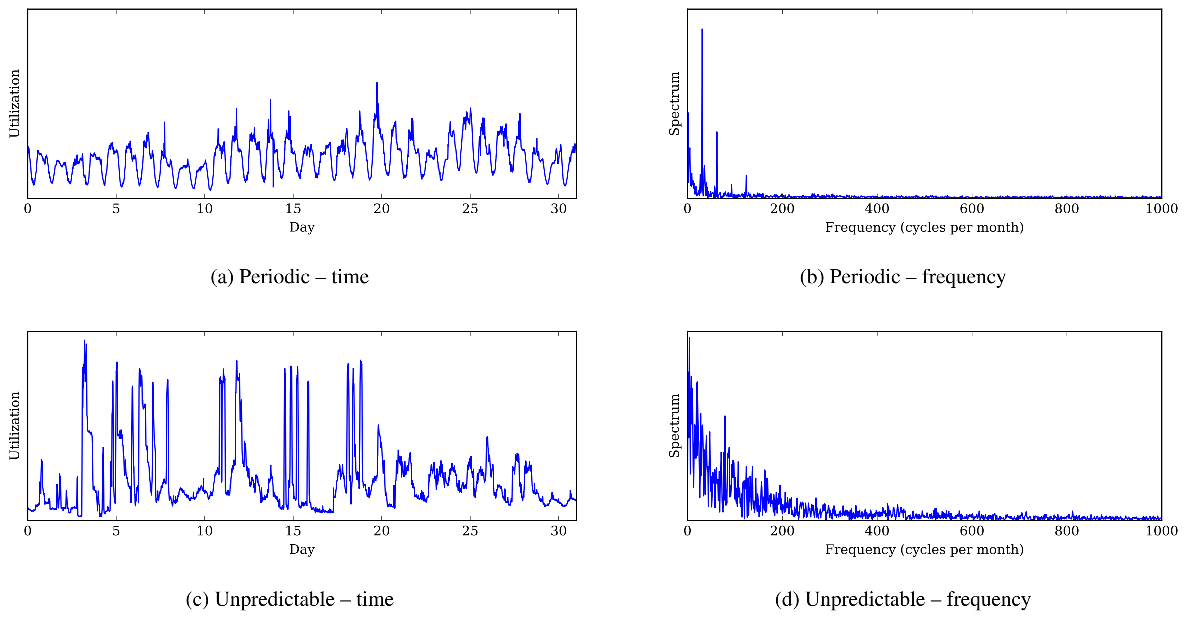
AutoPilot records the primary tenant utilization per server for all hardware resources, but for simplicity we focus on the CPU in this paper. It records the CPU utilization every two minutes. As the load is not always evenly balanced across all servers of a primary tenant, we compute the average of their utilizations in each time slot, and use the utilization of this *"average"* server for one month to represent the primary tenant.

We then identify trends in the tenants' utilizations, using signal processing. Specifically, we use the Fast Fourier Transform (FFT) on the data from each primary tenant individually. The FFT transforms the utilization time series into the frequency domain, making it easy to identify any periodicity (and its strength) in the series.

We identify three main classes of primary tenants: *periodic*, *unpredictable*, and (roughly) *constant*. Figure 1 shows the CPU utilization trends of a periodic and an unpredictable primary tenant in the time and frequency domains. Figure 1b shows a strong signal at frequency 31, because there are 31 days (load peaks and valleys) in that month. In contrast, Figure 1d shows a decreasing trend in signal strength as the frequency increases, as the majority of the signal derives from events that rarely happen (*i.e.*, exhibit lower frequency).

As one would expect, user-facing primary tenants often exhibit periodic utilization (*e.g.*, high during the day and low at night), whereas non-user-facing (*e.g.*, Web crawling, batch data analytics) or non-production (*e.g.*, development, testing) primary tenants often do not. For example, a Web crawling or data scrubber tenant may exhibit (roughly) constant utilization, whereas a testing tenant often exhibits unpredictable utilization behavior.

More interestingly, Figure 2 shows that user-facing (periodic) primary tenants are actually a small minority. The vast majority of primary tenants exhibit roughly constant CPU utilization. Nevertheless, Figure 3 shows that the periodic primary tenants represent a large percentage (~40% on average) of the servers in each datacenter.



(a) Periodic – time

(b) Periodic – frequency

(c) Unpredictable – time

(d) Unpredictable – frequency

Figure 1: Sample periodic and unpredictable one-month traces in the time and frequency domains.

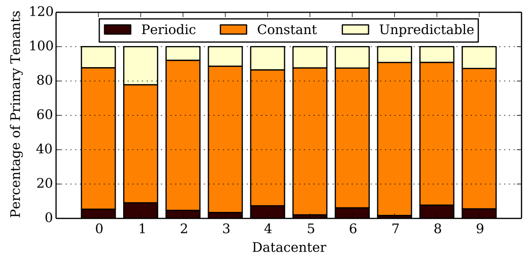


Figure 2: Percentages of primary tenants per class.

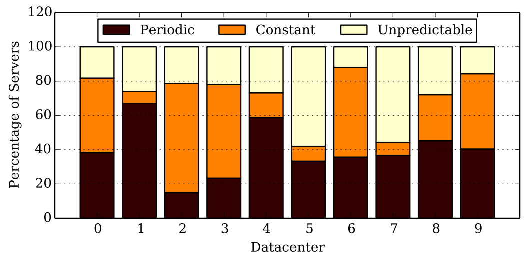


Figure 3: Percentages of servers per class.

Still, the non-periodic primary tenants account for more than half of the tenants and servers.

Most importantly, the vast majority of servers (~75%) run primary tenants (periodic and constant) for which the historical utilization data is a good predictor of future behaviors (the utilizations repeat periodically or all the time). Thus, leveraging this data should improve the quality of both our task scheduling and data placement.

3.3 Disk reimaging

Disk reimages are relatively frequent for some primary tenants, which by itself potentially threatens data durability under co-location. Even worse, disk reimages are often correlated, *i.e.* many servers might be reimaged at the same time (*e.g.*, when servers are repurposed from one primary tenant to another). Thus, it is critical for data durability to account for reimages and correlations.

AutoPilot collects disk reimaging (reformatting) data per server. This data includes reimages of multiple types: (1) those initiated manually by developers or service operators intending to re-deploy their environments (primary tenants) or re-start them from scratch; (2) those initiated by AutoPilot to test the resilience of production services; and (3) those initiated by AutoPilot when disks have undergone maintenance (*e.g.*, tested for failure).

We now study the reimaging patterns using three years of data from AutoPilot. As an example of the reimaging frequencies we observe, Figure 4 shows the Cumulative Distribution Function (CDF) of the average number of reimages per month for each server in three years in five representative datacenters in our sample. Figure 5 shows the CDF of the average number of reimages per server per month for each primary tenant for the same years and datacenters. The discontinuities in this figure are due to short-lived primary tenants.

We make three observations from these figures. First and most importantly, there is a good amount of diver-

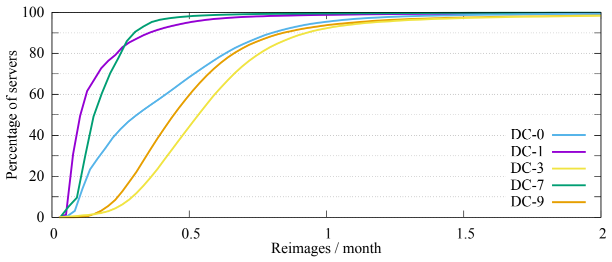


Figure 4: Per-server number of reimages in three years.

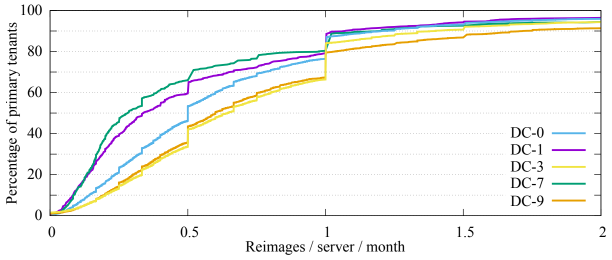


Figure 5: Per-tenant number of reimages in three years.

sity in average reimaging frequency across primary tenants in each datacenter (Figure 5 does not show nearly vertical lines). Second, the reimaging frequencies per month are fairly low in all datacenters. For example, at least 90% of servers are reimaged once or fewer times per month on average, whereas at least 80% of primary tenants are reimaged once or fewer times per server per month on average. This shows that reimaging by primary tenant engineers and AutoPilot is not overly aggressive on average, but there is a significant tail of servers (10%) and primary tenants (20%) that are reimaged relatively frequently. Third, the primary tenant reimaging behaviors are fairly consistent across datacenters, though three datacenters show substantially lower reimaging rates per server (we show two of those datacenters in Figure 4).

The remaining question is whether each primary tenant exhibits roughly the same frequencies month after month. In this respect, we find that there is substantial variation, as frequencies sometimes change substantially.

Nevertheless, when compared to each other, primary tenants tend to rank consistently in the same part of the spectrum. In other words, primary tenants that experience a relatively small (large) number of reimages in a month tend to experience a relatively small (large) number of reimages in the following month. To verify this trend, we split the primary tenants of a datacenter into three frequency groups, each with the same number of tenants: *infrequent*, *intermediate*, and *frequent*. Then, we track the movement of the primary tenants across these groups over time. Figure 6 plots the CDF of the number of times a primary tenant changed groups from one month to the next. At least 80% of primary tenants

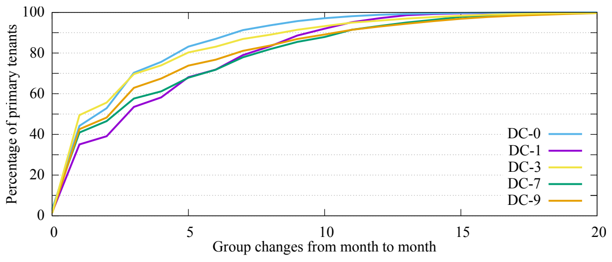


Figure 6: Number of times a primary tenant changed reimage frequency groups in three years.

changed groups only 8 or fewer times out of the possible 35 changes in three years. This behavior is also consistent across datacenters.

Again, these figures show that historical reimaging data should provide meaningful information about the future. Using this data should improve data placement.

4 Smart Co-location Techniques

In this section, we describe our techniques for smart task scheduling and data placement, which leverage the primary tenants' historical behavior patterns.

4.1 Smart task scheduling

We seek to schedule batch tasks (secondary tenants) to harvest spare cycles from servers that natively run interactive services and their supporting workloads (primary tenants). Modern cluster schedulers achieve high job performance and/or fairness, so they are good candidates for this use. However, their designs typically assume dedicated servers, *i.e.* there are no primary tenants running on the same servers. Thus, we must (1) modify them to become aware of the primary tenants and the primary tenants' priority over the servers' resources; and (2) endow them with scheduling algorithms that reduce the number of task killings resulting from the co-located primary tenants' need for resources. The first requirement is fairly easy to accomplish, so we describe our implementation in Section 5. Here, we focus on the second requirement, *i.e.* smart task scheduling, and use historical primary tenant utilization data to select servers that will most likely have the required resources available throughout the tasks' entire executions.

Due to the sheer number of primary tenants, it would be impractical to treat them independently during task scheduling. Thus, our scheduling technique first clusters together primary tenants that have similar utilization patterns into the same utilization *class*, and then select a class for the tasks of a job. Next, we discuss our clustering and class selection algorithms in turn.

Algorithm 1 Class selection algorithm.

```
1: Given: Classes $C$, Headroom($type$,$c$), Ranking Weights $W$
2: function SCHEDULE(Batch job $J$)
3:     $J$.type = Length (short, medium, or long) from its last run
4:     $J$.req = Max amount of concurrent resources from DAG
5:     for each $c \in C$ do
6:         $c$.weightedroom=Headroom($J$.type,$c$) $\times$ $W$[$J$.type,$c$.class]
7:     end for
8:     $F = \{\forall c \in C |$ Headroom($J$.type,$c$) $\geq J$.req$\}$
9:     if $F \neq \emptyset$ then
10:        Pick 1 class $c \in F$ probabilistically $\propto$ $c$.weightedroom
11:        return $\{c\}$
12:    else if Job $J$ can fit in multiple classes combined then
13:        Pick $\{c_0, \ldots, c_k\} \subseteq C$ probabilistically $\propto$ $c$.weightedroom
14:        return $\{c_0, \ldots, c_k\}$
15:    else
16:        Do not pick classes
17:        return $\{\emptyset\}$
18:    end if
19: end function
```

The **clustering** algorithm periodically (*e.g.*, once per day) takes the most recent time series of CPU utilizations from the average server of each primary tenant, runs the FFT algorithm on the series, groups the tenants into the three patterns described in Section 3 (periodic, constant, unpredictable) based on their frequency profiles, and then uses the K-Means algorithm to cluster the profiles in each pattern into classes. Clustering tags each class with the utilization pattern, its average utilization, and its peak utilization. It also maintains a mapping between the classes and their primary tenants.

As we detail in Algorithm 1, our **class selection** algorithm relies on the classes defined by the clustering algorithm. When we need to allocate resources for a job's tasks, the algorithm selects a class (or classes) according to the expected job length (line 3) and a predetermined ranking of classes for the length. We represent the desired ranking using weights (line 6); higher weight means higher ranking. For a long job, we give priority to constant classes first, then periodic classes, and finally unpredictable classes. We prioritize the constant classes in this case because constant-utilization primary tenants with enough available resources are unlikely to take resources away from the job during its execution. At the other extreme, a short job does not require an assurance of resource availability long into the future; knowing the current utilization is enough. Thus, for a short job, we rank the classes unpredictable first, then periodic, and finally constant. For a medium job, the ranking is periodic first, then constant, and finally unpredictable.

We categorize a job as short, medium, or long by comparing the duration of its last execution to two predefined thresholds (line 3). We set the thresholds based on the historical distribution of job lengths and the current computational capacity of each preferred tenant class (*e.g.*, the total computation required by long jobs should be proportional to the computational capacity of constant primary tenants). Importantly, the last duration need *not* be an accurate execution time estimate. Our goal is much easier: to categorize jobs into three rough types. We assume that a job that has not executed before is a medium job. After a possible error in this first guess, we find that a job consistently falls into the same type.

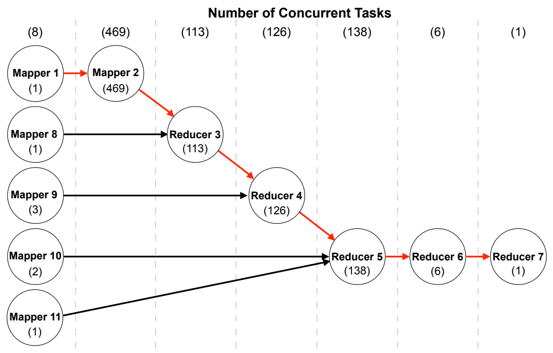


Figure 7: Example job execution DAG.

We estimate the maximum amount of concurrent resources that the job will need (line 4) using a breadth-first traversal of the job's directed acyclic graph (DAG), which is a common representation of execution flows in many frameworks [1, 29, 40]. We find this estimate to be accurate for our workloads. Figure 7 shows an example job DAG (query 19 from TPC-DS [34]), for which we estimate a maximum of 469 concurrent containers.

Whether a job "fits" in a class (line 8) depends on the amount of available resources (or the amount of *headroom*) that the servers in the class currently exhibit, as we define below. When multiple classes could host the job, the algorithm selects one with probability proportional to its weighted headroom (lines 9 and 10). If multiple classes are necessary, it selects as many classes as needed, again probabilistically (lines 12 and 13). If there are not enough resources available in any combination of classes, it does not select any class (line 16).

The headroom depends on the job type. For a short job, we define it as 1 minus the current average CPU utilization of the servers in the class. For a medium job, we use 1 minus Max(average CPU utilization, current CPU utilization). For a long job, we use 1 minus Max(peak CPU utilization, current CPU utilization).

4.2 Smart data placement

Modern distributed file systems achieve high data access performance, availability, and durability, so there is a strong incentive for using them in our harvesting scenario. However, like cluster schedulers, they assume dedicated servers without primary tenants running and storing data on the same servers, and without primary tenant owners deliberately reimaging disks. Thus, we

Algorithm 2 Replica placement algorithm.

```
1: Given: Storage space available in each server, Primary reimaging
2:        stats, Primary peak CPU util stats, Desired replication $R$
3: function PLACE REPLICAS(Block $B$)
4:     Cluster primary tenants wrt reimaging and peak CPU util
5:         into 9 classes, each with the same total space
6:     Select the class of the server creating the block
7:     Select the server creating the block for one replica
8:     for $r = 2$; $r \leq R$; $r = r + 1$ do
9:         Select the next class randomly under two constraints:
10:            No class in the same row has been picked
11:            No class in the same column has been picked
12:        Pick a random primary tenant of this class as long as
13:            its environment has not received a replica
14:        Pick a server in this primary tenant for the next replica
15:        if ($r$ mod 3) == 0 then
16:            Forget rows and columns that have been selected so far
17:        end if
18:    end for
19: end function
```

must (1) modify them to become co-location-aware; and (2) endow them with replica placement algorithms that improve data availability and durability in the face of primary tenants and how they are managed. Again, the first requirement is fairly easy to accomplish, so we discuss our implementation in Section 5. Here, we focus on the second requirement, *i.e.* smart replica placement.

The challenge is that the primary tenants and the management system may hurt data availability and durability for any block: (1) if the replicas of a block are stored in primary tenants that load-spike at the same time, the block may become unavailable; (2) if developers or the management system reimage the disks containing all the replicas of a block in a short time span, the block will be lost. A replica placement algorithm must then account for primary tenant and management system activity.

An intuitive best-first approach would be to try to find primary tenants that reimage their disks the least, and from these primary tenants select the ones that have lowest CPU utilizations. However, this greedy approach has two serious flaws. First, it treats durability and availability independently, one after the other, ignoring their interactions. Second, after the space at all the "good" primary tenants is exhausted, new replicas would have to be created at locations that would likely lead to poor durability, poor availability, or both.

We prefer to make decisions that promote durability and availability at the same time, while consistently spreading the replicas around as evenly as possible across all types of primary tenants. Thus, our replica placement algorithm (Algorithm 2) creates a two-dimensional clustering scheme, where one dimension corresponds to durability (disk reimages) and the other to availability (peak CPU utilization). It splits the two-dimensional space into 3×3 classes (infrequent, intermediate, and frequent reimages versus low, medium, and high peak utilizations), each of which has the same amount of available storage for harvesting $S/9$, where S is the total amount of currently available storage (lines 4 and 5). This idea can be applied to splits other than 3×3, as long as they provide enough primary tenant diversity.

The above approach tries to balance the available space across classes. However, perfect balancing may be impossible when primary tenants have widely different amounts of available space, and the file system starts to become full. The reason is that balancing space perfectly could require splitting a large primary tenant across two or more classes. We prevent this situation by selecting a single class for each tenant, to avoid hurting placement diversity. The side effect is that small primary tenants get filled more quickly, causing larger primary tenants to eventually become the only possible targets for the replicas. This effect can be eliminated by not filling the file system to the point that less than three primary tenants remain as possible targets for replicas. In essence, there is a tradeoff between space utilization and diversity. We discuss this tradeoff further in Section 7.

When a client creates a new block, our algorithm selects one class for each replica. The first class is that of the server creating the block; the algorithm places a replica at this server to promote locality (lines 6 and 7). If the desired replication is greater than 1, it repeatedly selects classes randomly, in such a way that no row or column of the two-dimensional space has two selections (lines 9, 10, and 11). It places a replica in (a randomly selected server of) a randomly selected primary tenant in this class, while ensuring that no two primary tenants in the same environment receive a replica (lines 12, 13, and 14). Finally, for a desired replication level larger than 3, it does extra rounds of selections. At the beginning of each round, it forgets the history of row and column selections from the previous round (lines 15, 16, and 17).

The environment constraint is the only aspect of our techniques that is AutoPilot-specific. However, the constraints generalize to any management system: avoid placing multiple replicas in any logical (*e.g.*, environment) or physical (*e.g.*, rack) server grouping that induces correlations in resource usage, reimaging, or failures.

Figure 8 shows an example of our clustering scheme and primary tenant selection, assuming all primary tenants have the same amount of available storage. The rows defining the peak utilization classes do not align, as we ensure that the available storage is the same in all classes.

5 System Implementations

We implement our techniques into YARN, Tez, and HDFS. Next, we overview these systems. Then, we describe our implementation guidelines and systems, called YARN-H, Tez-H, and HDFS-H ("-H" refers to history).

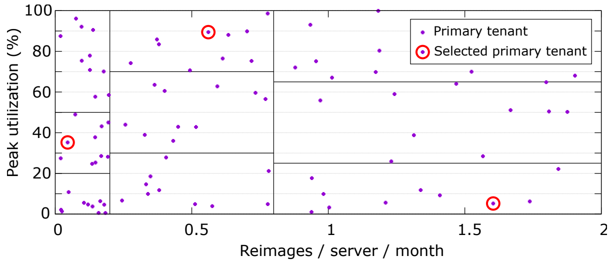


Figure 8: Two-dimensional clustering scheme.

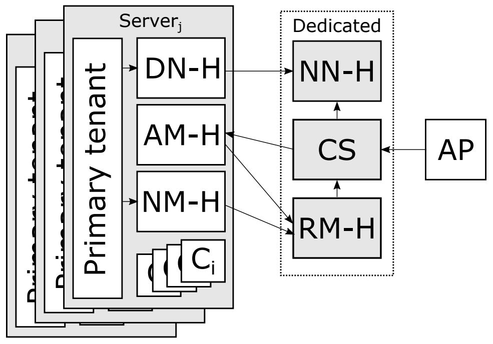


Figure 9: Overview of YARN-H (RM-H and NM-H), Tez-H (AM-H), and HDFS-H (NN-H and DN-H) in a co-location scenario. Our new clustering service (CS) interacts with all three systems. The arrows represent information flow. C_i = Container i; AP = AutoPilot.

5.1 Background

YARN [36] comprises a global Resource Manager (RM) running on a dedicated server, a Node Manager (NM) per server, and a per-job Application Master (AM) running on one of the servers. The RM arbitrates the use of resources (currently, cores and memory) across the cluster. The (primary) RM is often backed up by a secondary RM in case of failure. Each AM requests containers from the RM for running the tasks of its job. Each container request specifies the desired core and memory allocations for it, and optionally a "node label". The RM selects a destination server for each container that has the requested resources available and the same label. The AM decides which tasks it should execute in each container. The AM also tracks the tasks' execution, sequencing them appropriately, and re-starting any killed tasks. Each NM creates containers and reports the amount of locally available resources to the RM in periodic "heartbeats". The NM kills any container that tries to utilize more memory than its allocation.

Tez [29] is a popular framework upon which MapReduce, Hive, Pig, and other applications can be built. Tez provides an AM that executes complex jobs as DAGs.

HDFS [11] comprises a global Name Node (NN) running on a dedicated server, and a Data Node (DN) per server. The NN manages the namespace and the mapping of file blocks to DNs. The (primary) NN is typically backed up by a secondary NN. By default, the NN replicates each block (256 MBytes) three times: one replica in the server that created the block, one in another server of the same rack, and one in a remote rack. Upon a block access, the NN informs the client about the servers that store the block's replicas. The client then contacts the DN on any of these servers directly to complete the access. The DNs heartbeat to the NN; after a few missing heartbeats from a DN, the NN starts to re-create the corresponding replicas in other servers without overloading the network (30 blocks/hour/server).

System	Main extensions
YARN	Report primary tenant utilization to the RM
	Kill containers due to primary tenant needs
	Maintain resource reserve for primary tenant
	Probabilistically balance load
Tez	Leverage information on the observed job lengths
	Estimate max concurrent resource requirements
	Track primary tenant utilization patterns
	Schedule tasks on servers unlikely to kill them
	Schedule tasks on servers with similar primaries
HDFS	Track primary tenant utilization, deny accesses
	Report primary tenant status to the NN
	Exclude busy servers from info given to clients
	Track primary disk reimaging, peak utilizations
	Place replicas at servers with diverse patterns
General	Create dedicated environment for main components

Table 1: Our main extensions to YARN, Tez, and HDFS.

5.2 Implementation guidelines

We first must modify the systems to become aware of the primary tenants and their priority over the servers' resources. Because of this priority, we must ensure that the key components of these systems (RMs and NNs) do not share their servers with any primary tenants. Second, we want to integrate our history-based task scheduling and data placement algorithms into these systems.

Figure 9 overviews our systems. The arrows in the figure represent information flow. Each shared server receives one instance of our systems; other workloads are considered primary tenants. Table 1 overviews our main extensions. The next sections describe our systems.

5.3 YARN-H and Tez-H

Design goals: (G1) ensure that the primary tenant always gets the cores and memory it desires; (G2) ensure that there is always a reserve of resources for the primary tenant to spike into; and (G3) schedule the tasks on servers where they are less likely to be killed due to the resource needs of the corresponding primary tenants.

Primary tenant awareness. We implement goals G1 and G2 in YARN-H by modifying the NM to (1) track the primary tenant's core and memory *utilizations*; (2) round them up to the next integer number of cores and the next integer MB of memory; and (3) report the sum of these rounded values and the secondary tenants' core and memory *allocations* in its heartbeat to RM-H. If NM-H detects that there is no longer enough reserved resources, it replenishes the reserve back to the pre-defined amount by killing enough containers from youngest to oldest.
Smart task scheduling. We implement goal G3 by implementing a service that performs our clustering algorithm, and integrating our class selection algorithm into Tez-H. We described both algorithms in Section 4.1.

Tez-H requests the estimated maximum number of concurrent containers from RM-H. When Tez-H selects one class, the request names the node label for the class. When Tez-H selects multiple classes, it uses a disjunction expression naming the labels. RM-H schedules a container to a heartbeating server of the correct class with a probability proportional to the server's available resources. If Tez-H does not name a label, RM-H selects destination servers using its default policy.
Overheads. Our modifications introduce negligible overheads. For primary tenant awareness, we add a few system calls to the NM to get the resource utilizations, perform a few arithmetic operations, and piggyback the results to RM-H using the existing heartbeat. The clustering service works off the critical path of job execution, computes headrooms using a few arithmetic operations, and imposes very little load on RM-H. In comparison to its querying of RM-H once per minute, *every* server heartbeats to RM-H *every 3 seconds*. Tez-H requires a single interaction with the clustering service per job.

5.4 HDFS-H

Design goals: (G1) ensure that we never use more space at a server than allowed by its primary tenant; (G2) ensure that HDFS-H data accesses do not interfere with the primary tenant when it needs the server resources; and (G3) place the replicas of each block so that it will be as durable and available as possible, given the resource usage of the primary tenants and how they are managed.

Note that full data durability cannot be *guaranteed* when using harvested storage. For example, service engineers or the management system may reimage a large number of disks at the same time, destroying multiple replicas of a block. Obviously, one can increase durability by using more replicas. We explore this in Section 6.
Primary tenant awareness. For goal G1, we use an existing mechanism in HDFS: the primary tenants declare how much storage HDFS-H can use in each server.

Implementing goal G2 is more difficult. To make our changes seamless to clients, we modify the DN to deny data accesses when its replica is unavailable (*i.e.*, when allowing the access would consume some of the resource reserve), causing the client to try another replica. (If all replicas of a desired block are busy, the block becomes unavailable and Tez will fail the corresponding task.) In addition, DN-H reports being "busy" or available to NN-H in its heartbeats. If DN-H says that it is busy, NN-H stops listing it as a potential source for replicas (and stops using it as a destination for new replicas as well). When the CPU utilization goes below the reserve threshold, NN-H will again list the server as a source for replicas (and use it as a destination for new ones).
Smart replica placement. For goal G3, we integrate our replica placement algorithm (Section 4.2) into NN-H.
Overheads. Our extensions to HDFS impose negligible overheads. For primary tenant awareness, we add a few system calls to the DN to get the primary tenant CPU utilization, and piggyback the results to NN-H in the heartbeat. Denying a request under heavy load adds two network transfers, but this overhead is minimal compared to that of disk accesses. For smart replica placement, our modifications add the clustering algorithm to the NN, and the extra communication needed for it to receive the algorithm inputs. The clustering and data structure updates happen in the background, off the critical path.

6 Evaluation

6.1 Methodology

Experimental testbed. Our testbed is a 102-server setup, where each server has 12 cores and 32GB of memory. We reserve 4 cores (33%) and 10GB (31%) of memory for primary tenants to burst into based on empirical measurements of interference. (Recall that performance isolation technology at each server would enable smaller resource reserves.) To mimic realistic primary tenants, each server runs a copy of the Apache Lucene search engine [26], and uses more threads (up to 12) with higher load. We direct traffic to the servers to reproduce the CPU utilization of 21 primary tenants (13 periodic, 3 constant, and 5 unpredictable) from datacenter DC-9. We also reproduce the disk reimaging statistics of these primary tenants. For the batch workloads, we run 52 different Hive [33] queries (which translate into DAGs of relational processing tasks) from the TPC-DS benchmark [34]. We assume Poisson inter-arrival times (mean 300 seconds) for the queries.

We use multiple baselines. When studying scheduling, the first baseline is stock YARN and Tez. We call it *"YARN-Stock"*. The second baseline combines primary-tenant-aware YARN with stock Tez, but does not implement smart task scheduling. We call it *"YARN-PT"*.

We call our full system *"YARN-H/Tez-H"*. Given the workload above, we set the thresholds for distinguishing task length types to 173 and 433 seconds. Jobs shorter than 173 seconds are short, and longer than 433 seconds are long. These values produce resource requirements for the jobs of each type that roughly correspond to the amount of available capacity in the preferred primary tenant class for the type. We use HDFS-Stock with YARN-Stock, and HDFS-PT with the other YARN versions. The latter combination isolates the impact of primary tenant awareness in YARN from that in HDFS.

When studying data placement and access, the first baseline is *"HDFS-Stock"*, *i.e.* stock HDFS unaware of primary tenants. The second baseline is *"HDFS-PT"*, which brings primary tenant awareness to data accesses but does not implement smart data placement. We call our full system *"HDFS-H"*. We use YARN and Tez with HDFS-Stock, and YARN-PT and Tez with the other HDFS versions. Again, we seek to isolate the impact of primary tenant awareness in HDFS and YARN.

Simulator. Because we cannot experiment with entire datacenters and need to capture long-term behaviors (*e.g.*, months to years), we also built a simulator that reproduces the CPU utilization and reimaging behavior of all the primary tenants (thousands of servers) in the datacenters we study. We simulate servers of the same size and resource reserve as in our real experiments. To study a spectrum of utilizations, we also experiment with higher and lower traffic levels, each time multiplying the CPU utilization time series by a constant factor and saturating at 100%. Because of the inaccuracy introduced by saturation, we also study a method in which we scale the CPU utilizations using n^{th}-root functions (*e.g.*, square root, cube root). These functions make the higher utilizations change less than the lower ones when we scale them, reducing the chance of saturations.

When studying task scheduling and data availability, we simulate each datacenter for one month. When studying data durability, we simulate each datacenter for one year. We use the same set of Hive queries to drive our simulator, but multiply their lengths and container usage by a scaling factor to generate enough load for our large datacenters (many thousands of servers) while limiting the simulation time.

In the simulator, we use the same code that implements clustering, task scheduling, and data placement in our real systems. The simulator also reproduces key behaviors from the real systems, *e.g.* it reconstructs lost replicas at the same rate as our real HDFS systems. However, it does not model the primary tenants' response times. We compare our systems to the second baseline (YARN-PT) in task scheduling, and the first baseline (HDFS-Stock) in data placement and access.

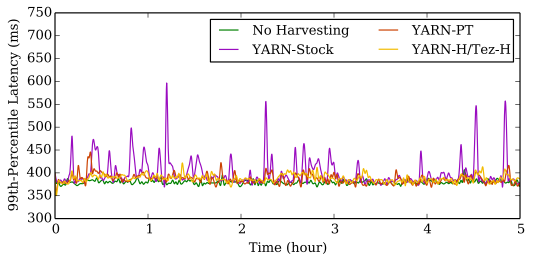


Figure 10: Primary tenant's tail latency in the real testbed for versions of YARN and Tez.

6.2 Performance microbenchmarks

The most expensive operations in our systems are the clustering and class selection in task scheduling and data placement. For task scheduling, clustering takes on average 2 minutes for the primary tenants of DC-9, when running single-threaded. (Recall that this clustering happens in the clustering service once per day, off the critical scheduling path.) The clustering produces 23 classes (13 periodic, 5 constant, and 5 unpredictable) for DC-9. For this datacenter, class selection takes less than 1 msec on average. For data placement, clustering and class selection take on average 2.55 msecs per new block (0.81 msecs in HDFS-Stock) for DC-9. (Clustering here can be done off the critical data placement path as well.)

6.3 Experimental results

Task scheduling comparisons. We start by investigating the impact of harvesting spare compute cycles on the performance of the primary tenant. Figure 10 shows the average of the servers' 99th-percentile response times (in ms) every minute during a five-hour experiment. The curve labeled "No Harvesting" depicts the tail latencies when we run Lucene in isolation. The other curves depict the Lucene tail latencies under different systems, when TPC-DS jobs harvest spare cycles across the cluster. The figure shows that YARN-Stock hurts tail latency significantly, as it disregards the primary tenant. In contrast, YARN-PT keeps tail latencies significantly lower and more consistent. The main reason is that YARN-PT actually kills tasks to ensure that the primary tenant's load can burst up without a latency penalty. Finally, YARN-H/Tez-H exhibits tail latencies that nearly match those of the No-Harvesting execution. The maximum tail latency difference is only 44 ms, which is commensurate with the amount of variance in the No-Harvesting execution (average tail latencies ranging from 369 to 406 ms). The improved tail latencies come from the more balanced utilization of the cluster capacity in YARN-H.

Another key characteristic of YARN-H/Tez-H is its smart scheduling of tasks to servers where they are less

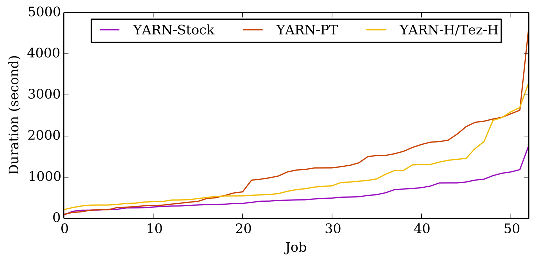


Figure 11: Secondary tenants' run times in the real testbed for versions of YARN and Tez.

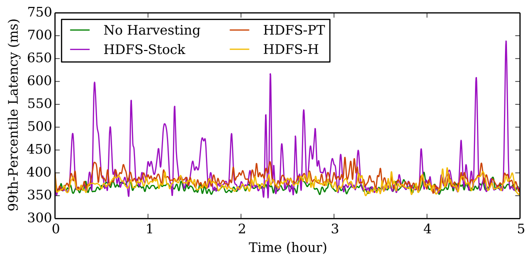


Figure 12: Primary tenant's tail latency in the real testbed for versions of HDFS.

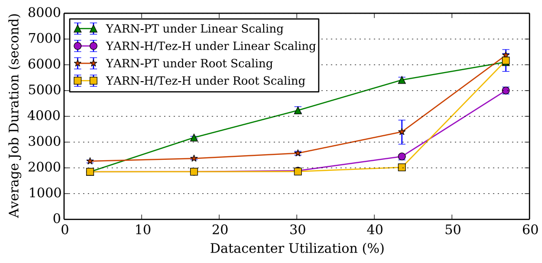


Figure 13: Secondary tenants' run time improvements in DC-9 under YARN-H/Tez-H for root and linear scalings.

likely to be killed. Figure 11 shows the execution times of all jobs in TPC-DS for YARN-Stock, YARN-PT, and YARN-H/Tez-H. As one would expect, YARN-Stock exhibits the lowest execution times. Unfortunately, this performance comes at the cost of ruining that of the primary tenant, which is unacceptable. Because YARN-PT must kill (and re-run) tasks when the primary tenant's load bursts, it exhibits substantially higher execution times, 1181 seconds on average. YARN-H/Tez-H lowers these times significantly to 938 seconds on average.

In these experiments, YARN-H/Tez-H improves the average CPU utilization from 33% to 54%, which is a significant improvement given that we reserve 33% of the CPU for primary tenant bursts. The utilization improvement depends on the utilization of the primary tenants (the lower their utilization, the more resources we can harvest), the resource demand coming from secondary tenants (the higher the demand, the more tasks we can schedule), and the resource reserve (the smaller the reserve, the more resources we can harvest).

Overall, these results clearly show that YARN-H/Tez-H is capable of both protecting primary tenant performance and increasing the performance of batch jobs.

Data placement and access comparisons. We now investigate whether HDFS-H is able to protect the performance of the primary tenant and provide higher data availability than its counterparts. Figure 12 depicts the average of the servers' 99th-percentile response times (in ms) every minute during another five-hour experiment. As expected, the figure shows that HDFS-Stock degrades tail latency significantly. HDFS-PT and HDFS-H reduce the degradation to at most 47 ms. The reason is that these versions avoid accessing/creating data at busy servers. However, HDFS-PT actually led to 47 failed accesses, *i.e.* these blocks could not be accessed as all of their replicas were busy. By using our smart data placement algorithm, HDFS-H eliminated all failed accesses.

6.4 Simulation results

Task scheduling comparisons. We start our simulation study by considering the full spectrum of CPU utilizations, assuming the size and behavior of our real production datacenters. Recall that we use two methods to scale utilizations (up and down) from the real utilizations: linear and root scalings. To isolate the benefit of our use of historical primary tenant utilizations, we compare YARN-H/Tez-H to YARN-PT. Figure 13 depicts the average batch job execution time in DC-9 under both systems and scalings, as a function of utilization. Each point along the curves shows the average of five runs, whereas the intervals range from the minimum average to the maximum average across the runs. As one would expect, high utilization causes higher queuing delays and longer execution times. (Recall that we reserve 33% of the resources for primary tenants to burst into, so queues are already long when we approach 60% utilization.) However, YARN-PT under linear scaling behaves differently; the average execution times start to increase significantly at lower utilizations. The reason is that linear scaling produces greater temporal variation in the CPU utilizations of each primary tenant than root scaling. Higher utilization variation means that YARN-PT is more likely to have to kill tasks, as it does not know the historical utilization patterns of the primary tenants. For example, at 45% utilization, YARN-PT under linear scaling kills 4× more tasks than the other system-scaling combinations.

Because YARN-H/Tez-H uses our clustering and smart task scheduling, it improves job performance significantly across most of the utilization spectrum. Under linear scaling, the average execution time reduction

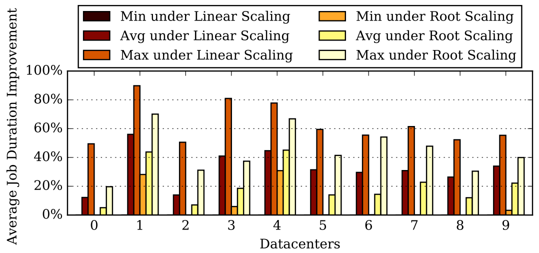


Figure 14: Secondary tenants' run time improvements from YARN-H/Tez-H for root and linear scalings.

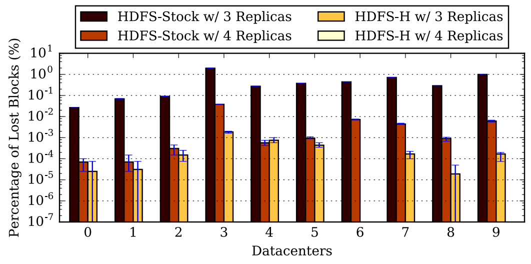


Figure 15: Lost blocks for two replication levels.

ranges from 0% to 55%, whereas under root scaling it ranges between 3% and 41%. The YARN-H/Tez-H advantage is larger under linear scaling, since the utilization pattern of each primary tenant varies more over time.

To see the impact of primary tenants with different characteristics than in DC-9, Figure 14 depicts the minimum, average, and maximum job execution time improvements from YARN-H/Tez-H across the utilization spectrum for each datacenter (five runs for each utilization level). The average improvements range from 12% to 56% under linear scaling, and 5% to 45% under root scaling. The lowest average improvements are for DC-0 and DC-2, which exhibit the least amount of primary tenant utilization variation over time. At the other extreme, the largest average improvements come for DC-1 and DC-4, as many of their primary tenants exhibit significant temporal utilization variations. The largest maximum improvements (~90% and ~70% under linear and root scaling, respectively) also come from these two datacenters, regardless of scaling type.

Data placement and access comparisons. We now consider the data durability in HDFS-H. Figure 15 shows the percentage of lost blocks under two replication levels (three and four replicas per block), as we simulate one year of reimages and 4M blocks. Each bar depicts the average of five runs, and the intervals range from the minimum to the maximum percent data loss in those simulations. The missing bars mean that there is no data loss in any of the corresponding five simulations. Note that a single lost block represents a 10^{-5} ($< 100 \times 1/4M$) percentage of lost blocks, *i.e.* 6 nines of durability.

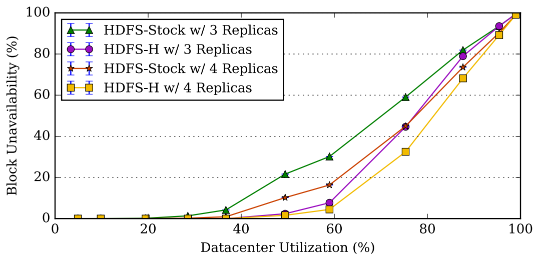


Figure 16: Failed accesses under linear scaling.

The figure shows that HDFS-H reduces data loss more than two orders of magnitude under three-way replication, compared to HDFS-Stock. Moreover, for one of the datacenters, HDFS-H eliminates all data loss under three-way replication. The maximum number of losses of HDFS-H in any datacenter was only 81 blocks (DC-3). Under four-way replication, HDFS-H completely eliminates data loss for all datacenters, whereas HDFS-Stock still exhibits losses across the board. These results show that our data placement algorithm provides significant improvements in durability, despite the harvested nature of the disk space and the relatively high reimage rate for many primary tenants. In fact, the losses with HDFS-H and three-way replication are lower than those with HDFS-Stock and four-way replication for all but one datacenter; *i.e.* our algorithm almost always achieves higher durability at a lower space overhead than HDFS-Stock.

Our data availability results are also positive. Figure 16 depicts the percentage of failed accesses under the two replication levels and linear scaling, as a function of the average utilization. The figure includes range bars from five runs, but they are all too small to see. The figure shows that HDFS-H exhibits no data unavailability up to higher utilizations (~40%) than HDFS-Stock, and low unavailability for even higher utilization (50%), under both replication levels. At 50% utilization, HDFS-Stock already exhibits relatively high unavailability under both replication levels. Around 66% utilization, unavailability starts to increase faster (accesses cannot proceed if CPU utilization is higher than 66%). More interestingly, our smart data placement under three-way replication achieves lower unavailability than HDFS-Stock under four-way replication below 75% utilization. The trends are similar under root scaling, except that HDFS-H exhibits no unavailability up to a higher utilization (50%) than with linear scaling. Regardless of the scaling type, HDFS-H can achieve higher availability at a lower space overhead than HDFS-Stock for most utilizations.

7 Experiences in Production

As a first rollout stage, we deployed HDFS-H to a production cluster with thousands of servers eleven months ago. Since then, we have been enabling/adding features as our deployment grows. For example, we extended the set of placement constraints beyond environments to include machine functions and physical racks. In addition, we initially configured the system to treat the replica placement constraints as "soft", *e.g.* the placement algorithm would allow multiple replicas in the same environment, to prevent the block creation from failing when the available space was becoming scarce. This initial decision promoted space utilization over diversity. Section 4.2 discusses this tradeoff.

Since its production deployment, our system has eliminated all data losses, except for a small number of losses due to corner-case bugs or promoting space over diversity. Due to the latter losses, we started promoting diversity over space utilization more than nine months ago. Since then, we have not lost blocks. For comparison, when the stock HDFS policy was activated by mistake in this cluster for just three days during this period, dozens of blocks were lost.

We also deployed YARN-H's primary tenant awareness code to production fourteen months ago, and have not experienced any issues with it (other than needing to fix a few small bugs). We are now productizing our scheduling algorithm and will deploy it to production.

In the process of devising, productizing, deploying, and operating our systems, we learned many lessons.

1. Even well-tested open-source systems require additional hardening in production. We had to create watchdogs that monitor key components of our systems to detect unavailability and failures. Because of the non-trivial probability of concurrent failures, we increased the number of RMs and NNs to four instead of two. Finally, we introduced extensive telemetry to simplify debugging and operation. For example, we collect extensive information about HDFS-H blocks to estimate its placement quality.

2. Synchronous operations and unavailability. Synchronous operations are inadequate when resources or other systems become unavailable. For example, our production deployments interact with a performance isolation manager (similar to [24]). This interaction was unexpectedly harmful to HDFS-H. The reason is that the manager throttles the secondary tenants' disk activity when the primary tenant performs substantial disk I/O. This caused the DN heartbeats on these servers to stop flowing, as the heartbeat thread does synchronous I/O to get the status of modified blocks and free space. As a result, the NN started a replication storm for data that it thought was lost. We then changed the heartbeat thread to become asynchronous and report the status that it most recently found.

3. Data durability is king. As we mention above, our initial HDFS-H deployment favored space over diversity, which caused blocks to be lost and the affected users to become quite exercised. By default, we now monitor the quality of placements and stop consuming more space when diversity becomes low. To recover some space, we still favor space usage over diversity for those files that do not have strict durability requirements.

4. Complexity is your enemy. As others have suggested [6], simplicity, modularity, and maintainability are highly valued in large production systems, especially as engineering teams change and systems evolve. For example, our initial task scheduling technique was more complex than described in Section 4.1. We had to simplify it, while retaining most of the expected gains.

5. Scaling resource harvesting to massive datacenters requires additional infrastructure. Stock YARN and HDFS are typically used in relatively small clusters (less than 4k servers), due to their centralized structure and the need to process heartbeats from all servers. Our goal is to deploy our systems to much larger installations, so we are now in the process of creating an implementation of HDFS-H that federates multiple smaller clusters and automatically moves files/folders across them based on primary tenant behaviors, and our algorithm's ability to provide high data availability and durability.

6. Contributing to the open-source community. Though our techniques are general, some of the code we introduced in our systems was tied to our deployments. This posed challenges when contributing changes to and staying in-sync with their open-source versions. For example, some of the YARN-H primary tenant awareness changes we made to Hadoop version 2.6 were difficult to port to version 2.7. Based on this experience, we refactored our code to isolate the most basic and general functionality, which we could then contribute back; some of these changes will appear in version 2.8.

8 Conclusion

In this paper, we first characterized all servers of ten large-scale datacenters. Then, we introduced techniques and systems that effectively harvest spare compute cycles and storage space from datacenters for batch workloads. Our systems embody knowledge of the existing primary workloads, and leverage historical utilization and management information about them. Our results from an experimental testbed and from simulations of the ten datacenters showed that our systems eliminate data loss and unavailability in many scenarios, while protecting primary workloads and significantly improving batch job performance. Based on these results, we conclude that our systems in general, and our task scheduling and data placement policies in particular, should enable datacenter operators to increase utilization and reduce TCO.

Acknowledgments

We thank our shepherd, Michael Stumm, for his help in improving the paper and his patience with us. We also thank Paulo Tomita, Sekhar Pasupuleti, Robert Grandl, and Srikanth Kandula for their help with our experimental setup. We are indebted to Karthik Kambatla for his help with open-sourcing some of our changes to YARN. We are also indebted to Sriram Rao, Carlo Curino, Chris Douglas, Vivek Narasayya, Manoj Syamala, Sameh Elnikety, Thomas F. Wenisch, and Willy Zwaenepoel for our many discussions about this work and their comments to our paper. Finally, we thank Gaurav Sareen and Eric Boyd for their support of this project.

References

[1] M. Abadi, P. Barham, J. Chen, Z. Chen, A. Davis, J. Dean, M. Devin, S. Ghemawat, G. Irving, M. Isard, M. Kudlur, J. Levenberg, R. Monga, S. Moore, D. G. Murray, B. Steiner, P. Tucker, V. Vasudevan, P. Warden, M. Wicke, Y. Yu, and X. Zheng. TensorFlow: A System for Large-Scale Machine Learning. In *Proceedings of the 12th USENIX Symposium on Operating System Design and Implementation*, 2016.

[2] L. Abraham, J. Allen, O. Barykin, V. Borkar, B. Chopra, C. Gerea, D. Merl, J. Metzler, D. Reiss, S. Subramanian, J. L. Wiener, and O. Zed. Scuba: Diving into Data at Facebook. *Proceedings of the VLDB Endowment*, 2013.

[3] F. Ahmad, S. T. Chakradhar, A. Raghunathan, and T. N. Vijaykumar. Tarazu: Optimizing MapReduce on Heterogeneous Clusters. In *Proceedings of the 17th International Conference on Architectural Support for Programming Languages and Operating Systems*, 2012.

[4] L. A. Barroso, J. Clidaras, and U. Hölzle. The Datacenter as a Computer: An Introduction to the Design of Warehouse-Scale Machines. *Synthesis Lectures on Computer Architecture*, 2013.

[5] M. Carvalho, W. Cirne, F. Brasileiro, and J. Wilkes. Long-term SLOs for Reclaimed Cloud Computing Resources. In *Proceedings of the ACM Symposium on Cloud Computing*, 2014.

[6] 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 Transactions on Computer Systems*, 2008.

[7] R. B. Clay, Z. Shen, and X. Ma. Accelerating Batch Analytics With Residual Resources From Interactive Clouds. In *Proceedings of the 21st International Symposium on Modelling, Analysis and Simulation of Computer and Telecommunication Systems*, 2013.

[8] C. Curino, D. E. Difallah, C. Douglas, S. Krishnan, R. Ramakrishnan, and S. Rao. Reservation-based Scheduling: If You'Re Late Don'T Blame Us! In *Proceedings of the ACM Symposium on Cloud Computing*, 2014.

[9] C. Delimitrou and C. Kozyrakis. Paragon: QoS-Aware Scheduling for Heterogeneous Datacenters. In *Proceedings of the 18th International Conference on Architectural Support for Programming Languages and Operating Systems*, 2013.

[10] C. Delimitrou and C. Kozyrakis. Quasar: Resource-Efficient and QoS-Aware Cluster Management. In *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems*, 2014.

[11] A. Foundation. HDFS Architecture Guide, 2008.

[12] A. Goder, A. Spiridonov, and Y. Wang. Bistro: Scheduling Data-Parallel Jobs Against Live Production Systems. In *Proceedings of the USENIX Annual Technical Conference*, 2015.

[13] I. Goiri, R. Bianchini, S. Nagarakatte, and T. D. Nguyen. ApproxHadoop: Bringing Approximations to MapReduce Frameworks. In *Proceedings of the 20th International Conference on Architectural Support for Programming Languages and Operating Systems*, 2015.

[14] I. Goiri, K. Le, T. D. Nguyen, J. Guitart, J. Torres, and R. Bianchini. GreenHadoop: Leveraging Green Energy in Data-processing Frameworks. In *Proceedings of the 7th ACM European Conference on Computer Systems*, 2012.

[15] R. Grandl, G. Ananthanarayanan, S. Kandula, S. Rao, and A. Akella. Multi-Resource Packing for Cluster Schedulers. In *Proceedings of the 2014 ACM SIGCOMM Conference*, 2014.

[16] 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*, 2011.

[17] M. Isard. Autopilot: Automatic Data Center Management. *SIGOPS Operating Systems Review*, 2007.

[18] K. Karanasos, S. Rao, C. Curino, C. Douglas, K. Chaliparambil, G. M. Fumarola, S. Heddaya, R. Ramakrishnan, and S. Sakalanaga. Mercury: Hybrid Centralized and Distributed Scheduling in Large Shared Clusters. In *Proceedings of the USENIX Annual Technical Conference*, 2015.

[19] H. Kasture and D. Sanchez. Ubik: Efficient Cache Sharing with Strict Qos for Latency-Critical Workloads. In *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems*, 2014.

[20] M. A. Laurenzano, Y. Zhang, L. Tang, and J. Mars. Protean Code: Achieving Near-Free Online Code Transformations for Warehouse Scale Computers. In *Proceedings of the 47th Annual IEEE/ACM International Symposium on Microarchitecture*, 2014.

[21] J. Leverich and C. Kozyrakis. Reconciling High Server Utilization and Sub-Millisecond Quality-of-Service. In *Proceedings of the 9th European Conference on Computer Systems*, 2014.

[22] H. Lin, X. Ma, J. Archuleta, W.-C. Feng, M. Gardner, and Z. Zhang. MOON: MapReduce On Opportunistic eNvironments. In *Proceedings of the 19th ACM International Symposium on High Performance Distributed Computing*, 2010.

[23] M. J. Litzkow, M. Livny, and M. W. Mutka. Condor-A Hunter of Idle Workstations. In *Proceedings of the 8th International Conference on Distributed Computing Systems*, 1988.

[24] D. Lo, L. Cheng, R. Govindaraju, P. Ranganathan, and C. Kozyrakis. Heracles: Improving Resource Efficiency at Scale. In *Proceedings of the 42nd Annual International Symposium on Computer Architecture*, 2015.

[25] J. Mars, L. Tang, R. Hundt, K. Skadron, and M. L. Soffa. Bubble-Up: Increasing Utilization in Modern Warehouse Scale Computers via Sensible Co-locations. In *Proceedings of the 44th Annual IEEE/ACM International Symposium on Microarchitecture*, 2011.

[26] M. McCandless, E. Hatcher, and O. Gospodnetic. *Lucene in Action: Covers Apache Lucene 3.0*. Manning Publications Co., 2010.

[27] D. Novakovic, N. Vasic, S. Novakovic, D. Kostic, and R. Bianchini. DeepDive: Transparently Identifying and Managing Performance Interference in Virtualized Environments. In *Proceedings of the USENIX Annual Technical Conference*, 2013.

[28] G. Ren, E. Tune, T. Moseley, Y. Shi, S. Rus, and R. Hundt. Google-Wide Profiling: A Continuous Profiling Infrastructure for Data Centers. *IEEE Micro*, 2010.

[29] B. Saha, H. Shah, S. Seth, G. Vijayaraghavan, A. Murthy, and C. Curino. Apache Tez: A Unifying Framework for Modeling and Building Data Processing Applications. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, 2015.

[30] B. Sharma, T. Wood, and C. R. Das. HybridMR: A Hierarchical MapReduce Scheduler for Hybrid Data Centers. In *Proceedings of the 33rd International Conference on Distributed Computing Systems*, 2013.

[31] L. Tang, J. Mars, and M. L. Soffa. Compiling for Niceness: Mitigating Contention for QoS in Warehouse Scale Computers. In *Proceedings of the 10th International Symposium on Code Generation and Optimization*, 2012.

[32] L. Tang, J. Mars, W. Wang, T. Dey, and M. L. Soffa. ReQoS: Reactive Static/Dynamic Compilation for QoS in Warehouse Scale Computers. In *Proceedings of the 18th International Conference on Architectural Support for Programming Languages and Operating Systems*, 2013.

[33] A. Thusoo, J. S. Sarma, N. Jain, Z. Shao, P. Chakka, S. Anthony, H. Liu, P. Wyckoff, and R. Murthy. Hive: A Warehousing Solution Over a Map-Reduce Framework. *Proceedings of the VLDB Endowment*, 2009.

[34] Transaction Processing Performance Council. TPC Benchmarks.

[35] D. Tsafrir, Y. Etsion, and D. G. Feitelson. Backfilling Using System-Generated Predictions Rather Than User Runtime Estimates. *IEEE Transactions on Parallel and Distributed Systems*, 2007.

[36] 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*, 2013.

[37] A. Verma, L. Pedrosa, M. Korupolu, D. Oppenheimer, E. Tune, and J. Wilkes. Large-scale Cluster Management at Google with Borg. In *Proceedings of the 10th European Conference on Computer Systems*, 2015.

[38] H. Yang, A. Breslow, J. Mars, and L. Tang. Bubble-flux: Precise Online QoS Management for Increased Utilization in Warehouse Scale Computers. In *Proceedings of the 40th Annual International Symposium on Computer Architecture*, 2013.

[39] M. Zaharia, D. Borthakur, J. Sen Sarma, K. Elmeleegy, S. Shenker, and I. Stoica. Delay Scheduling: A Simple Technique for Achieving Locality and Fairness in Cluster Scheduling. In *Proceedings of the 5th European Conference on Computer Systems*, 2010.

[40] M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker, and I. Stoica. Spark: Cluster Computing with Working Sets. In *Proceedings of the 2nd USENIX Workshop on Hot Topics in Cloud Computing*, 2010.

[41] W. Zhang, S. Rajasekaran, S. Duan, T. Wood, and M. Zhuy. Minimizing Interference and Maximizing Progress for Hadoop Virtual Machines. *SIGMETRICS Performance Evaluation Review*, 2015.

[42] X. Zhang, E. Tune, R. Hagmann, R. Jnagal, V. Gokhale, and J. Wilkes. CPI2: CPU Performance Isolation for Shared Compute Clusters. In *Proceedings of the 8th ACM European Conference on Computer Systems*, 2013.

[43] Y. Zhang, M. A. Laurenzano, J. Mars, and L. Tang. SMiTe: Precise QoS Prediction on Real-System SMT Processors to Improve Utilization in Warehouse Scale Computers. In *Proceedings of the 47th Annual IEEE/ACM International Symposium on Microarchitecture*, 2014.

DQBarge: Improving data-quality tradeoffs in large-scale Internet services

Michael Chow[*], *Kaushik Veeraraghavan*[†], *Michael Cafarella*[*], *and Jason Flinn*[*],
University of Michigan[*] *Facebook, Inc.*[†]

Abstract

Modern Internet services often involve hundreds of distinct software components cooperating to handle a single user request. Each component must balance the competing goals of minimizing service response time and maximizing the quality of the service provided. This leads to low-level components making *data-quality tradeoffs*, which we define to be explicit decisions to return lower-fidelity data in order to improve response time or minimize resource usage.

We first perform a comprehensive study of low-level data-quality tradeoffs at Facebook. We find that such tradeoffs are widespread. We also find that existing data-quality tradeoffs are often suboptimal because the low-level components making the tradeoffs lack global knowledge that could enable better decisions. Finally, we find that most tradeoffs are reactive, rather than proactive, and so waste resources and fail to mitigate system overload.

Next, we develop DQBarge, a system that enables better data-quality tradeoffs by propagating critical information along the causal path of request processing. This information includes data provenance, load metrics, and critical path predictions. DQBarge generates performance and quality models that help low-level components make better, more proactive, tradeoffs. Our evaluation shows that DQBarge helps Internet services mitigate load spikes, improve utilization of spare resources, and implement dynamic capacity planning.

1 Introduction

A *data-quality tradeoff* is an explicit decision by a software component to return lower-fidelity data in order to improve response time or minimize resource usage. Data-quality tradeoffs are often found in Internet services due to the need to balance the competing goals of minimizing the service response time perceived by the end user and maximizing the quality of the service provided. Tradeoffs in large-scale services are pervasive since hundreds or thousands of distinct software components may be invoked to service a single request and each component may make individual data-quality tradeoffs.

Data-quality tradeoffs in low-level software components often arise from defensive programming. A programmer or team responsible for a specific component wishes to bound the response time of their component even when the resource usage or latency of a sub-service is unpredictable. For instance, a common practice is to time out when a sub-service is slow to respond and supply a default value in lieu of the requested data.

To quantify the prevalence of data-quality tradeoffs, we undertake a systematic study of software components at Facebook. We find that over 90% of components perform data-quality tradeoffs instead of failing. Some tradeoffs we observe are using default values, calculating aggregates from a subset of input values, and retrieving alternate values from a stale or lower-quality data source. Further, we observe that the vast majority of data-quality tradeoffs are reactive rather than proactive, e.g., components typically set timeouts and make data-quality tradeoffs when timers expires rather than predict which actions can be performed within a desired time bound.

These existing data-quality tradeoffs are suboptimal for three reasons. First, they consider only local knowledge available to the low-level software component because of the difficulty in accessing higher-level knowledge such as the provenance of data, system load, and whether the component is on the critical request path. Second, the tradeoffs are usually reactive (e.g., happening only after a timeout) rather than proactive (e.g., issuing only the amount of sub-service requests that can be expected to complete within a time bound); reactive tradeoffs waste resources and exacerbate system overload. Finally, there is no mechanism to trace the set of data-quality tradeoffs made during a request, and this makes understanding the quality and performance impact of such tradeoffs on actual requests difficult.

DQBarge addresses these problems by propagating critical information along the causal path of request processing. The propagated data includes load metrics, as well as the expected critical path and slack for individual software components. It also includes provenance for request data such as the data sources queried and the software components that have transformed the data. Finally, it includes the specific data-quality tradeoffs that have been made for each request; e.g., which data values were left out of aggregations.

In an offline stage, DQBarge uses this data to generate performance and quality models for low-level tradeoffs in the service pipeline. Later, while handling production traffic, it consults the models to proactively determine which tradeoffs to make.

DQBarge generates performance and quality models by sampling a small percentage of the total requests processed by the service and redundantly executing them to compare the performance and quality when different tradeoffs are employed. Redundant execution minimizes interference with production traffic; duplicated requests run offline on execution pipelines dedicated to model generation. Performance models capture how throughput and latency are affected by specific data-quality tradeoffs as a factor of overall system load and provenance. Quality models capture how the fidelity of the final response is affected by specific tradeoffs as a function of input data provenance.

These models enable better tradeoffs during the processing of subsequent production requests. For each production request, DQBarge passes extra data along the causal path of request processing. It predicts the critical path for each request and which software components will have substantial slack in processing time. It also measures current system load. This global and request-specific state is attached to the request at ingress. As the request propagates through software components, DQBarge annotates data objects with provenance. This information and the generated models are propagated to the low-level components, enabling them to make better tradeoffs.

We investigate three scenarios in which better data-quality tradeoffs can help. First, during unanticipated load spikes, making better data quality tradeoffs can maintain end-to-end latency goals while minimizing the loss in fidelity perceived by users. Second, when load levels permit, components with slack in their completion time can improve the fidelity of the response without impacting end-to-end latency. Finally, understanding the potential effects of low-level data-quality tradeoffs can inform dynamic capacity planning and maximize utility as a function of the resources required to produce output.

One way to frame this work is that data-quality tradeoffs are a specific type of quality-of-service tradeoff [7, 25, 29], akin to recent work in approximate computing [4, 8, 19, 18, 28, 30]. The distinguishing feature of data-quality tradeoffs is that they are embedded in low-level software components within complex Internet pipelines. This leads to a lack of global knowledge and makes it difficult for individual components to determine how making specific tradeoffs will impact overall service latency and quality. DQBarge addresses this issue by incorporating principles from the literature on causal tracing [5, 9, 10, 13, 23, 26, 27, 31] to propagate needed knowledge along the path of request processing, enabling better tradeoffs by providing the ability to assess the impact of tradeoffs.

Thus, this work makes the following contributions. First, we provide the first comprehensive study of low-level data-quality tradeoffs in a large-scale Internet service. Second, we observe that causal propagation of request statistics and provenance enables better and more proactive data-quality tradeoffs. Finally, we demonstrate the feasibility of this approach by designing, implementing, and evaluating DQBarge, an end-to-end approach for tracing, modeling, and actuating data-quality tradeoffs in Internet service pipelines.

We have added a complete, end-to-end implementation of DQBarge to Sirius [15], an open-source, personal digital assistant service. We have also implemented and evaluated the main components of the DQBarge architecture at Facebook and validated them with production data. Our results show that DQBarge can meet latency goals during load spikes, utilize spare resources without impacting end-to-end latency, and maximize utility by dynamically adjusting capacity for a service.

2 Study of data-quality tradeoffs

In this section, we quantify the prevalence and type of data-quality tradeoffs in production software at Facebook. We perform a comprehensive study of Facebook client services that use an internal key-value store called *Laser*. *Laser* enables online accessing of the results of a batch offline computation such as a Hive [33] query.

We chose to study clients of *Laser* for several reasons. First, *Laser* had 463 client services, giving us a broad base of software to examine. We systematically include all 463 services in our study to gain a representative picture of how often data-quality tradeoffs are employed at Facebook. Second, many details about timeouts and tradeoffs are specified in client-specific RPC configuration files for this store. We processed these files automatically, which reduced the amount of manual code inspection required for the study. Finally, we believe a key-value store is representative of the low-level components employed by most large-scale Internet companies.

Table 1 shows the results of our study for the 50 client services that invoke *Laser* most frequently, and Table 2 shows results for all 463 client services. We categorize how clients make data-quality decisions along two dimensions: proactivity and resultant action. Each entry shows the number of clients that make at least one data quality decision with a specific proactivity/action combination. For most clients, all decisions fall into a single category. A few clients use different strategies at different points in their code. We list these clients in multiple categories, so the total number of values in each table is slightly more than the number of client services.

	Failure	Data-quality tradeoff		
		Default	Omit	Alternate
Reactive	5 (10%)	14 (28%)	30 (60%)	1 (2%)
Proactive	0 (0%)	0 (0%)	2 (4%)	1 (2%)

Table 1: Data-quality decisions of the top 50 *Laser* clients. Each box shows the number of clients that make decisions according to the specified combination of reactive/proactive determination and resultant action. The total number of values is greater than 50 since a few clients use more than one strategy.

	Failure	Data-quality tradeoff		
		Default	Omit	Alternate
Reactive	40 (9%)	250 (54%)	174 (38%)	4 (1%)
Proactive	0 (0%)	3 (1%)	7 (2%)	1 (0%)

Table 2: Data-quality decisions made by all *Laser* clients. Each box shows the number of clients that make tradeoffs according to the specified combination of reactive/proactive determination and resultant action. The total number of values is greater than 463 since a few clients use more than one strategy.

2.1 Proactivity

We consider a tradeoff to be *reactive* if the client service always initiates the request and then uses a timeout or return code to determine if the request is taking too long or consuming too many resources. For instance, we observed many latency-sensitive clients that set a strict timeout for how long to wait for a response. If *Laser* takes longer than the timeout, such clients make a data-quality tradeoff or return a failure.

A *proactive* check predicts whether the expected latency or resource cost of processing the request will exceed a threshold. If so, a data-quality tradeoff is made immediately without issuing the request. For example, we observed a client that determines whether or not a query will require cross-data-center communication because such communication would cause it to exceed its latency bound. If there are no hosts that can service the query in its data center, it makes a data-quality tradeoff.

2.2 Resultant actions

We also examine the actions taken in response to latency or resource usage exceeding a threshold. *Failure* shows the number of clients that require a response from *Laser*. If the store responds with an error or timeout, the client fails. Such instances mean a programmer has chosen to not make a data-quality tradeoff.

The remaining categories represent different types of data-quality tradeoffs. *Default* shows the number of clients that return a pre-defined default answer when a tradeoff is made. For instance, we observed a client service that ranks chat threads according to their activity level. The set of most active chat groups are retrieved from *Laser* and boosted to the top of a chat bar. If retrieving this set fails or times out, chat groups and contacts are listed alphabetically.

The *Omit* category is common in clients that aggregate hundreds of values from different sources; e.g., to generate a model. If an error or timeout occurs retrieving values from one of these sources, those values are left out and the aggregation is performed over the values that were retrieved successfully.

One example we observed is a recommendation engine that aggregates candidates and features from several data sources. It is resilient to missing candidates and features. Although missing candidates are excluded from the final recommendation and missing features negatively affect candidate scores in calculating the recommendation, the exclusion of a portion of these values allows a usable but slightly lower-fidelity recommendation to be returned in a timely manner in the event of failure or unexpected system load.

The *Alternate* category denotes clients that make a tradeoff by retrieving an alternate, reduced quality, value from a different data source. For example, we observed a client that requests a pre-computed list of top videos for a given user. If a timeout or failure occurs retrieving this list, the client retrieves a more generic set of videos for that user. As a further example, we observed a client that chooses among a pre-ranked list of optimal data sources. On error or timeout, the client retrieves the data from the next best data source. This process continues until a response is received.

Before performing our study, we hypothesized that client services might try to retrieve data of equal fidelity from an alternate data store in response to a failure. However, we did not observe any instance of this behavior in our study (all alternate sources had lower-fidelity data).

2.3 Discussion of results

Tables 1 and 2 show that data quality tradeoffs are pervasive in the client services we study. 90% of the top 50 *Laser* clients and 91% of all 463 clients perform a data-quality tradeoff in response to a failure or timeout; the remaining 9-10% of clients consider the failure to retrieve data in a timely manner to be a fatal error. Thus, in the Facebook environment, making data-quality tradeoffs is normal behavior, and failures are the exception.

For the top 50 clients, the most common action when faced with a failure or timeout is to omit the requested value from the calculation of an aggregate (60%). The next most common action (28%) is to use a default value in lieu of the requested data. These trends are reversed when considering all clients. Only 36% of all 463 clients omit the requested values from an aggregation, whereas 52% use a default value.

We were surprised that only a few clients react to failure or timeout by attempting to retrieve the requested data from an alternate source (4% of the top 50 clients and 1% of all clients). This may be due to tight time or resource constraints; e.g., if the original query takes too long, there may be no time left to initiate another query.

Only 6% of the top 50 clients and 2% of all clients are proactive. The lack of proactivity represents a significant lost opportunity for optimization because requests that timeout or fail consume resources but produce no benefit. This effect can be especially prominent when requests are failing due to excessive load; a proactive strategy would decrease the overall stress on the system. When a proactive check fails, the service performing that check always makes a data-quality tradeoff (as opposed to terminating its processing with a failure); it would be very pessimistic for a client to return a failure without at least attempting to fetch the needed data.

In our inspection of source code, we observed that low-level data-quality decisions are almost always encapsulated within clients and not reported to higher-level components or attached to the response data. Thus, there is no easy way for operators to check how the quality of the response being sent to the user has been impacted by low-level quality tradeoffs during request processing.

3 Design and implementation

Motivated by our study results, we designed DQBarge to help developers understand the impact of data-quality tradeoffs and make better, more proactive tradeoffs to improve quality and performance. Our hypothesis is that propagating additional information along the causal path of request processing will provide the additional context necessary to reach these goals.

DQBarge has two stages of operation. During the offline stage, it samples a small percentage of production requests, and it runs a copy of each sampled request on a duplicate execution pipeline. It perturbs these requests by making specific data quality tradeoffs and measuring the request latency and result quality. DQBarge generates performance and quality models by systematically sweeping through the space of varying request load and data provenance dimensions specified by the developer and using multidimension linear regression over the data to predict performance and quality as a factor of load and provenance. Note that because requests are duplicated, end users are not affected by the perturbations required to gather model data. Further, DQBarge minimizes interference with production traffic by using dedicated resources for running duplicate requests as much as possible.

During the online stage, DQBarge uses the quality and performance models to decide when to make data-quality tradeoffs for production traffic in order to realize a configured goal such as maximizing quality subject to a specified latency constraint. It gathers the inputs to the models (load levels, critical path predictions, and provenance of data) and propagates them along the critical path of request execution by embedding the data in RPC objects associated with the request. At each potential tradeoff site, the low-level component calls DQBarge. DQBarge performs a model lookup to determine whether to make a data-quality tradeoff, and, if so, the specific tradeoff to make (e.g., which values to leave out of an aggregation). The software service then makes these tradeoffs proactively. DQBarge can optionally log the decisions that are made so that developers can understand how they are affecting production results.

The separation of work into online and offline stages is designed to minimize overhead for production traffic. These stages can run simultaneously; DQBarge can generate a new model offline by duplicating requests while simultaneously using an older model to determine what tradeoffs to make for production traffic. The downside of this design is that DQBarge will not react immediately to environmental changes outside the model parameters such as a code update that modifies resource usage. Instead, such changes will be reflected only after a new model is generated. We therefore envision that models are regenerated regularly (e.g., every day) or after significant environmental changes occur (e.g., after a major push of new code).

Section 3.1 describes how DQBarge gathers and propagates data about request processing, including system load, critical path and slack predictions, data provenance, and a history of the tradeoffs made during request processing. This data gathering and propagation is used by both the online and offline stages. Section 3.2 relates how DQBarge duplicates the execution of a small sample of requests for the offline stage and builds models of performance and quality for potential data-quality tradeoffs. As described in Section 3.3, DQBarge uses these models during the online stage to make better tradeoffs for subsequent requests: it makes proactive tradeoffs to reduce resource wastage, and it uses provenance to choose tradeoffs that lead to better quality at a reduced performance cost. Finally, Section 3.4 describes how DQBarge logs all tradeoffs made during request processing so that operators can review how system performance and request quality have been impacted.

3.1 Data gathering and propagation

DQBarge provides a library for developers to specify the information that should be propagated along the critical path. The library is implemented in 3268 lines of C++ code, plus Java bindings for services implemented in that language. Developers use the library interface to annotate objects during request processing and query those annotations at later stages of the pipeline. Table 3 shows selected functions from the DQBarge library API to which we will refer in the following discussion.

The DQBarge library has a RPC-package-specific back-end that modifies and queries existing RPC objects

DQBarge API
`putMetric(scope, key, type, value)`
`getMetric(key) → (type, value)`
`addProvenance(data object, key, type, value)`
`removeProvenance(data object, key)`
`getProvenance(data object) → list <key, type, value>`
`makeAggregationTradeoff(performance model, quality model, list<key>, list<object>) → list<object>`

Table 3: Selected functions from the DQBarge API

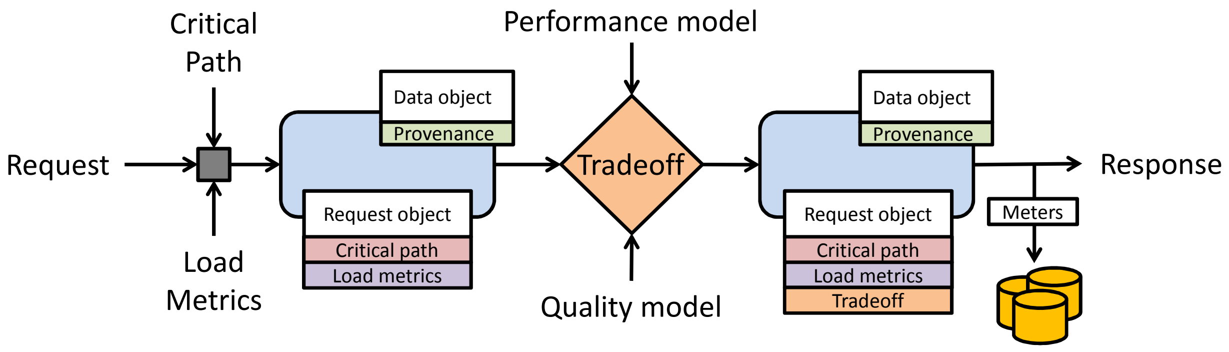


Figure 1: DQBarge overview.

to propagate the information. It modifies RPC objects by adding additional fields that contain data to be propagated along the causal path. It supports three object scopes: request-level, component-level, and data-level.

Request-level data are passed through all components involved in processing the request, following the causal path of request execution; such data includes system-wide load metrics, slack predictions, and a list of actual data-quality tradeoffs made during execution of the particular request. Services call `putMetric` to add this data to the request, specifying `request` as the scope and a typed key-value pair to track. Later, they may retrieve the data by calling `getMetric` with the specified key. The services in our case studies both have a global object containing a unique request identifier; DQBarge appends request-level information to this object. This technique for passing and propagating information is widely used in other tracing systems that follow the causal path of execution [13, 23].

Component-level objects persist from the beginning to end of processing for a specific software component within the request pipeline. Such objects are passed to all sub-components that are called during the execution of the higher-level component. DQBarge appends component-specific data to these objects, so such data will be automatically deallocated when execution passes beyond the specified component. Component-specific load metrics are one example of such data. To add this data, services call `putMetric` and specify a component-level RPC object as the scope.

Data-level objects are the specific data items being propagated as a result of request execution. `addProvenance` associates a typed key-value pair with a specific data object, since the provenance is meaningful only as long as the data object exists. A data object may have multiple provenance values.

Our library provide a useful interface for manipulating RPC objects, but developers must still make domain-specific decisions, e.g., what metrics and provenance values to add, what objects to associate with those values, and what rules to use to model the propagation of provenance. For instance, to reflect the flow of provenance in a component, developers should call `getProvenance` to retrieve the provenance of the inputs and `addProvenance` and `removeProvenance` to show causal propagation to outputs. Figure 1 shows an overview of how this data propagates through the system.

Load metrics may be relevant to the entire request or only to certain components. Each load metric is a typed key-value pair (e.g., a floating point value associated with the key "requests/second"). Currently supported load metrics are throughput, CPU load, and memory usage.

Critical path and slack predictions are specified as directed acyclic graphs. Each software component in the graph has a weight that corresponds to its predicted slack (the amount of additional time it could take to process a request without affecting the end-to-end latency of the request). Components on the critical path of request execution have zero slack. DQBarge relies on an external component, the Mystery Machine, to make critical path and slack predictions; [11] describes the details of that system. Currently, slack predictions are made at request ingress; such predictions may cover the entire request or only specific components of the request. The graphs for our two case studies in Section 4 are relatively small, so we transmit this data by value along the request processing path (using `PutMetric` and a graph type). If we were to deploy DQBarge along the entire Facebook request processing path, then the graphs would be much larger, and we would likely need to transmit them by reference or only send relevant subgraphs to components.

DQBarge associates provenance with the data objects it describes. Provenance can be a data source or the algorithm employed to generate a particular object. Provenance is represented as an unordered collection of typed key-value pairs. DQBarge supports both discrete and continuous types. DQBarge extracts a schema for the quality model from the data objects passed to tradeoff functions such as `makeAggregationTradeoff` by iterating through all provenance entries attached to each object to read the provenance keys and their associated types. Components are treated as black boxes, so developers must specify how provenance is propagated when a component modifies existing data objects or creates new ones.

Finally, DQBarge stores the tradeoffs that were made during request processing in a request-level object. As described in Section 3.4, this information may be logged and used for reporting the effect of tradeoffs on quality and performance.

3.2 Model generation

For each potential tradeoff, DQBarge creates a performance model and a quality model that capture how the tradeoff affects request execution. Performance models predict how throughput and latency are affected by specific data-quality tradeoffs as a factor of overall system load and the provenance of input data. Quality models capture how the fidelity of the final response is affected by specific tradeoffs as a function of provenance.

DQBarge uses *request duplication* to generate models from production traffic without adversely affecting the user experience. At the RPC layer, it randomly samples incoming requests from production traffic, and it routes a copy of the selected requests to one or more request duplication pipelines. Such pipelines execute isolated, redundant copies of the request for which DQBarge can make different data-quality tradeoffs. These pipelines do not return results to the end user and they are prevented from making modifications to persistent stores in the production environment; in all other respects, request execution is identical to production systems. Many production systems, including those at Facebook, already have similar functionality for testing purposes, so adding support for model generation required minimal code changes.

DQBarge controls the rate at which requests enter the duplication pipeline by changing the sampling frequency. At each potential tradeoff site, services query DQBarge to determine which tradeoffs to make; DQBarge uses these hooks to systematically explore different tradeoff combinations and generate models. For instance, `makeAggregationTradeoff` specifies a point where values can be omitted from an aggregation; this function returns a list of values to omit (an empty list means no tradeoff). DQBarge has similar functions for each type of tradeoff identified in Section 2.

To generate a performance model, DQBarge uses load testing [20, 24]. Each data-quality tradeoff offers multiple fidelities. A default value may be used or not. Different types or percentages of values can be left out of an aggregation. Multiple alternate data stores may be used. For each fidelity, DQBarge starts with a low request rate and increases the request rate until the latency exceeds a threshold. Thus, the resulting model shows request processing latency as a function of request rate and tradeoffs made (i.e., the fidelity of the tradeoff selected). DQBarge also records the provenance of the input data for making the tradeoff; the distribution of provenance is representative of production traffic since the requests in the duplication pipeline are a random sampling of that traffic. DQBarge determines whether the resulting latency distribution varies as a result of the input provenance; if so, it generates separate models for each provenance category. However, in the systems we study in Section 4, provenance does not have a statistically significant effect on performance (though it does significantly affect quality).

Quality models capture how the fidelity of the final response is affected by data-quality tradeoffs during request processing. To generate a quality model, DQBarge sends each request to two duplication pipelines. The first pipeline makes no tradeoffs, and so produces a full-fidelity response. The second pipeline makes a specified tradeoff, and so produces a potentially lower-fidelity response. DQBarge measures the quality impact of the tradeoff by comparing the two responses and applying a service-specific quality ranking specified by the developer. For example, if the output of the request is a ranked list of Web pages, then a service-specific quality metric might be the distance between where pages appear in the two rankings.

DQBarge next learns a model of how provenance affects request quality. As described in the previous section, input data objects to the component making the tradeoff are annotated with provenance in the form of typed key-value pairs. These pairs are the features in the quality model. DQBarge generates observations by making tradeoffs for objects with different provenance; e.g., systematically using default values for different types of objects. DQBarge uses multidimension linear regression to model the importance of each provenance feature in the quality of the request result. For example, if a data-quality tradeoff omits values from an aggregation, then omitting values from one data source may have less impact than omitting values from a different source.

Provenance can substantially reduce the number of observations needed to generate a quality model. Recall that all RPC data objects are annotated with provenance; thus, the objects in the final request result have provenance data. In many cases, the provenance relationship is direct; an output object depends only on a specific input provenance. In such cases, we can infer that the effect of a data-quality tradeoff would be to omit the specified output object, replace it with a default value, etc. Thus, given a specific output annotated with provenance, we can infer what the quality would be if further tradeoffs were made (e.g., a specific set of provenance features were used to omit objects from an aggregation). In such cases, the processing of one request can generate many data points for the quality model. If the provenance relationship is not direct, DQBarge generates these data points by sampling more requests and making different tradeoffs.

3.3 Using the models

DQBarge uses its performance and quality models to make better, more proactive data-quality tradeoffs. System operators specify a high-level goal such as maximizing quality given a latency cap on request processing. Components call functions such as `makeAggregationTradeoff` at each potential tradeoff point during request processing; DQBarge returns a decision as to whether a tradeoff should be made and, if appropriate, what fidelity should be employed (e.g., which data source to use or which values to leave out of an aggregation). Services provide a reference to the performance and quality models, as well as a list of load metrics (identified by key) and identifiers for objects with provenance. The service then implements the tradeoff decision proactively; i.e., it makes the tradeoff immediately. This design does not preclude reactive tradeoffs. An unexpectedly delayed response may still lead to a timeout and result in a data-quality tradeoff.

DQBarge currently supports three high-level goals: maximizing quality subject to a latency constraint, maximizing quality using slack execution time available during request processing, and maximizing utility as a function of quality and performance. These goals are useful for mitigating load spikes, efficiently using spare resources, and implementing dynamic capacity planning, respectively. We next describe these three goals.

3.3.1 Load Spikes

Services are provisioned to handle peak request loads. However, changes in usage or traffic are unpredictable; e.g., the launch of a new feature may introduce additional traffic. Thus, systems are designed to handle unexpected load spikes; the reactive data-quality tradeoffs we saw in Section 2 are one such mechanism. DQBarge improves on existing practice by letting an operator specify a maximum latency for a request or a component of request processing. It maximizes quality subject to this constraint by making data-quality tradeoffs.

At each tradeoff site, there may be many potential tradeoffs that can be made (e.g., sets of values with different provenance may be left out of an aggregation or distinct alternate data stores may be queried). DQBarge orders possible tradeoffs by “bang for the buck” and greedily selects tradeoffs until the latency goal is reached. It ranks each potential tradeoff by the ratio of the projected improvement in latency (given by the performance model) to the decrease in request fidelity (given by the quality model). The independent parameters of the models are the current system load and the provenance of the input data. DQBarge selects tradeoffs in descending order of this ratio until the performance model predicts that the latency limit will be met.

3.3.2 Utilizing spare resources

DQBarge obtains a prediction of which components are on the critical path and which components have slack available from the Mystery Machine [11]. If a component has slack, DQBarge can make tradeoffs that improve quality without negatively impacting the end-to-end request latency observed by the user. Similar to the previous scenario, DQBarge calculates the ratio of quality improvement to latency decrease for each potential tradeoff (the difference is that this goal involves improving quality rather than performance). It greedily selects tradeoffs according to this order until the additional latency would exceed the projected slack time.

3.3.3 Dynamic capacity planning

DQBarge allows operators to specify the utility (e.g., the dollar value) of reducing latency and improving quality. It then selects the tradeoffs that improve utility until no more such tradeoffs are available. DQBarge also allows operators to specify the impact of adding or removing resources (e.g., compute nodes) as a utility function parameter. DQBarge compares the value of the maximum utility function with more and less resources and generates a callback if adding or removing resources would improve the current utility. Such callbacks allow dynamic re-provisioning. Since DQBarge uses multidimension linear regression, it will not model significantly non-linear relationships in quality or performance; more sophisticated learning methods could be used in such cases.

3.4 Logging data-quality decisions

DQBarge optionally logs all data-quality decisions and includes them in the provenance of the request data objects. The information logged includes the software component, the point in the execution where a tradeoff decision was made, and the specific decision that was made (e.g., which values were left out of an aggregation). To reduce the amount of data that is logged, only instances where a tradeoff was made are recorded. Timeouts and error return codes are also logged if they result in a reactive data-quality tradeoff. This information helps system administrators and developers understand how low-level data-quality tradeoffs are affecting the performance and quality of production request processing.

3.5 Discussion

DQBarge does not guarantee an optimal solution since it employs greedy algorithms to search through potential tradeoffs. However, an optimal solution is likely unnecessary given the inevitable noise that arises from predicting traffic and from errors in modeling. For the last use case, DQBarge assumes that developers can quantify the impact of changes to service response times, quality, and the utilization of additional resources in order to set appropriate goals. DQBarge also assumes that tradeoffs are independent, since calculating models over joint distributions would be difficult. Finally, because DQBarge compares quality across different executions of the same request with different tradeoffs, it assumes that request processing is mostly deterministic.

Using DQBarge requires a reasonably-detailed understanding of the service being modified. Developers must identify points in the code where data-quality tradeoffs should be made. They must specify what performance and quality metrics are important to their service. Finally, they must select which provenance values to track and specify how these values are propagated through black-box components. For both of the case studies in Section 4, a single developer who was initially unfamiliar with the service being modified was able to add all needed modifications, and these modifications comprised less than 450 lines of code in each case.

DQBarge works best for large-scale services. Although it generates models offline to reduce interference with production traffic, model generation does consume extra resources through duplication of request processing. For large Internet services like Facebook, the extra resource usage is a tiny percentage of that consumed by production traffic. However, for a small service that sees only a few requests per minute, the extra resources needed to generate the model may not be justified by the improvement in production traffic processing.

4 Case studies

We have implemented the main components of DQBarge in a portion of the Facebook request processing pipeline, and we have evaluated the results using Facebook production traffic. Our current Facebook implementation allows us to track provenance, generate performance and quality models and measure the efficacy of the data-quality tradeoffs available through these models. This implementation thus allows us to understand the feasibility and potential benefit of applying these ideas to current production code.

We have also implemented the complete DQBarge system in Sirius [15], an open-source personal digital assistant akin to Siri. Our Sirius implementation enables end-to-end evaluation of DQBarge, such as observing how data-quality tradeoffs can be used to react to traffic spikes and the availability of slack in the request pipeline.

4.1 Facebook

Our implementation of DQBarge at Facebook focuses on a page ranking service, which we will call *Ranker* in this paper. When a user loads the Facebook home page, *Ranker* uses various parameters of the request, such as the identity of the requester, to generate a ranked list of page recommendations. *Ranker* first generates candidate recommendations. It has a flexible architecture that allows the creation and use of multiple candidate generators; each generator is a specific algorithm for identifying possible recommendations. At the time of our study, there were over 30 generators that collectively produced hundreds of possible recommendations for each request.

Ranker retrieves feature vectors for each candidate from *Laser*, the key-value store we studied in Section 2. *Ranker* is a service that makes reactive data-quality tradeoffs. If an error or timeout occurs when retrieving features, *Ranker* omits the candidate(s) associated with those features from the aggregation of candidates and features considered by the rest of the *Ranker* pipeline.

Ranker uses the features to calculate a score for each candidate. The algorithm for calculating the score was opaque to us (it is based on a machine learning model regenerated daily). It then orders candidate by score and returns the top N candidates.

DQBarge leverages existing tracing and monitoring infrastructure at Facebook. It uses a production version of the Mystery Machine tracing and performance analysis infrastructure [11]. This tool discovers and reports performance characteristics of the processing of Facebook requests, including which components are on the critical path. From this data, we can calculate the slack available for each component of request processing; prior results have shown that, given an observation of past requests by the same user, slack for future requests can be predicted with high accuracy. Existing Facebook systems monitor load at each component in the pipeline.

DQBarge annotates data passed along the pipeline with provenance. The data object for each candidate is annotated with the generator that produced the data.

Similarly, features and other data retrieved for each candidate are associated with their data source.

We implemented meters at the end of the *Ranker* pipeline that measure the latency and quality of the final response. To measure quality, we compare the difference in ranking of the top N pages returned from the full-quality response (with no data-quality tradeoffs made) and the lower-fidelity response (that includes some tradeoffs). For example, if the highest-ranked page in the lower-fidelity response is the third-ranked page in the full-quality response, the *quality drop* is two.

4.2 Sirius

We also applied DQBarge to Sirius [15], an open-source personal assistant similar to Apple's Siri or Google Now. Sirius answers fact-based questions based on a set of configurable data sources. The default source is an indexed Wikipedia database; an operator may add other sources such as online search engines.

Sirius generates several queries from a question; each query represents a unique method of parsing the question. For each query, it generates a list of documents that are relevant to answering the query. Each document is passed through a natural language processing pipeline to derive possible answers. Sirius assigns each answer a numerical score and returns the top-ranked answer.

Data-quality tradeoffs in Sirius occur when aggregating values from multiple sub-service queries. Our DQBarge implementation makes these tradeoffs proactively by using quality and performance models to decide which documents to leave out of the aggregation when the system is under load.

Initially, Sirius did not have request tracing or load monitoring infrastructure. We therefore added the ability to trace requests and predict slack by adding the Mystery Machine to Sirius. For load, we added counters at each pipeline stage to measure request rates. Additionally, we track the CPU load and memory usage of the entire service. The performance data, predicted slack, and load information are all propagated by DQBarge as each request flows through the Sirius pipeline.

In each stage of the Sirius pipeline, provenance is propagated along with data objects. For example, when queries are formed from the original question, the algorithm used to generate the query is associated with the query object. Sirius provenance also includes the data used to generate the list of candidate documents.

Since Sirius did not have a request duplication mechanism, we added the ability to sample requests and send the same request through multiple instances of the Sirius pipeline. User requests are read-only with respect to Sirius data stores, so we did not have to isolate any modifications to service state from duplicated requests.

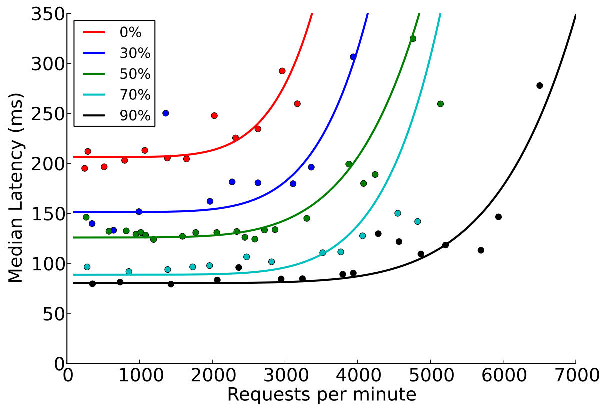


Figure 2: *Ranker* performance model This graph shows the effect of varying the frequency of data-quality tradeoffs on *Ranker* request latency. We varied the request rate by sampling different percentages of live production traffic at Facebook.

5 Evaluation

Our evaluation answers the following questions:

- Do data-quality tradeoffs improve performance?
- How much does provenance improve tradeoffs?
- How much does proactivity improve tradeoffs?
- How well does DQBarge meet end-to-end performance and quality goals?

5.1 Experimental setup

For *Ranker*, we perform our evaluation on Facebook servers using live Facebook traffic by sampling and duplicating *Ranker* requests. Our entire implementation uses duplicate pipelines, so as to not affect the results returned to Facebook users. Each pipeline duplicates traffic to a single isolated front-end server that is identical to those used in production. The duplicate pipelines share services from production back-end servers, e.g., those hosting key-value stores, but they are a small percentage of the total load seen by such servers. We change the load within a pipeline by sampling a larger or smaller number of *Ranker* requests and redirecting the sampled requests to a single front-end server for the pipeline.

For Sirius, we evaluated our end-to-end implementation of DQBarge on 16-core 3.1 GHz Xeon servers with 96 GB of memory. We send Sirius questions sampled from an archive from previous TREC conferences [32].

5.2 Performance benefits

We first measure the effect of data-quality tradeoffs on throughput and latency by generating performance models for *Ranker* and Sirius; Section 5.3 considers the effect of these tradeoffs on quality. DQBarge performs a full parameter sweep through the dimensions of request rate, tradeoff frequency, and provenance of the data being considered for each tradeoff, sampling at regular intervals. For brevity, we report a portion of these results.

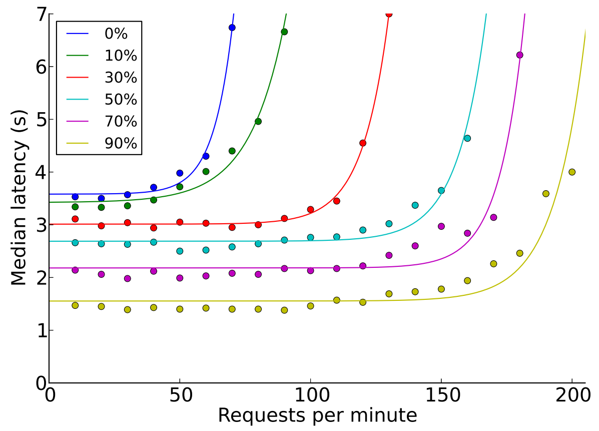


Figure 3: Sirius performance model. This graph shows the effect of varying the frequency of data-quality tradeoffs on Sirius request latency. Each curve shows a different tradeoff rate.

We show the median response time calculated over the sampling period at a specified request rate and tradeoff rate. For Sirius, 900 requests were sent over the sampling period. Median response time is shown because it is used for the remainder of the evaluation.

5.2.1 *Ranker*

Figure 2 shows the latency-response curve for *Ranker* when DQBarge varies the incoming request rate. Each curve shows the best fit for samples taken at a different *tradeoff rate*, which we define to be the object-level frequency at which data tradeoffs are actually made. When making tradeoffs, *Ranker* omits objects from aggregations; thus, to achieve a target tradeoff rate of x% during model generation, DQBarge will instruct *Ranker* to drop x% of the specific candidates. At a tradeoff rate of 0%, no candidates are dropped.

These results show that data-quality tradeoffs substantially improve *Ranker* latency at low loads (less than 2500 requests/minute); e.g., at a 30% tradeoff rate, median latency decreases by 28% and latency of requests in the 99th percentile decreases by 30%. Prior work has shown that server slack at Facebook is predictable on a per-request basis [11]. Thus, *Ranker* could make more tradeoffs to reduce end-to-end response time when *Ranker* is on the critical path of request processing, yet it could still provide full-fidelity responses when it has slack time for further processing.

Data-quality tradeoffs also improve scalability under load. Taking 250 ms as a reasonable knee in the latency-response curve, *Ranker* can process approximately 2500 requests per minute without making tradeoffs, but it can handle 4300 requests per minute when the tradeoff rate is 50% (a 72% increase). This allows *Ranker* to run at a lower fidelity during a load spike.

DQBarge found that the provenance of the data values selected for tradeoffs does not significantly affect performance. In other words, while the number of tradeoffs made has the effect shown in Figure 2, the specific candidates that are proactively omitted from an aggregation do not matter. Thus, we only show the effect of the request rate and tradeoff rate.

5.2.2 Sirius

Figure 3 shows results for Sirius. Like *Ranker*, the provenance of the data items selected for tradeoffs did not affect performance, so we show latency-response curves that vary both request rate and tradeoff rate.

The results for Sirius are similar to those for *Ranker*. A tradeoff rate of 50% reduces median end-to-end request latency by 26% and the latency of requests in the 99th percentile by 38%. Under load, a 50% tradeoff rate increases Sirius throughput by approximately 200%.

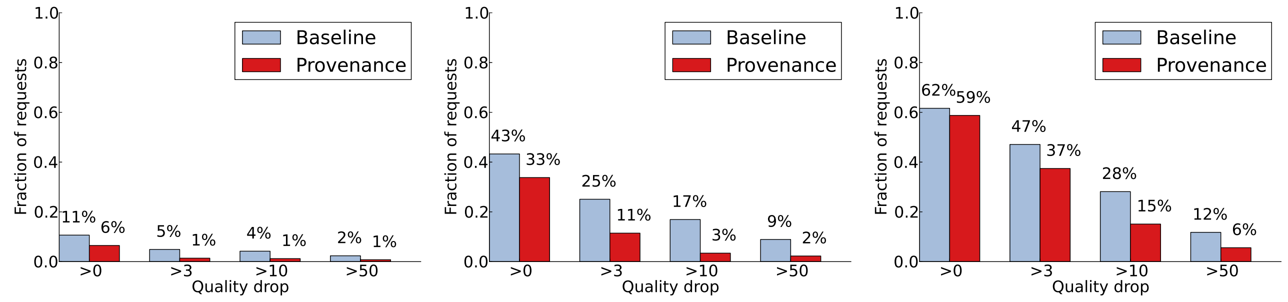
5.3 Effect of provenance

We next consider how much provenance improves the tradeoffs made by DQBarge. We consider a baseline quality model that does not take into account any provenance; e.g., given a target tradeoff rate, it randomly omits data values from an aggregation. This is essentially the policy in existing systems like *Ranker* and Sirius because there is no inherent order in requests from lower-level services to data stores; thus, timeouts affect a random sampling of the values returned. In contrast, DQBarge uses its quality model to select which values to omit, with the objective of choosing those that affect the final output the least.

5.3.1 *Ranker*

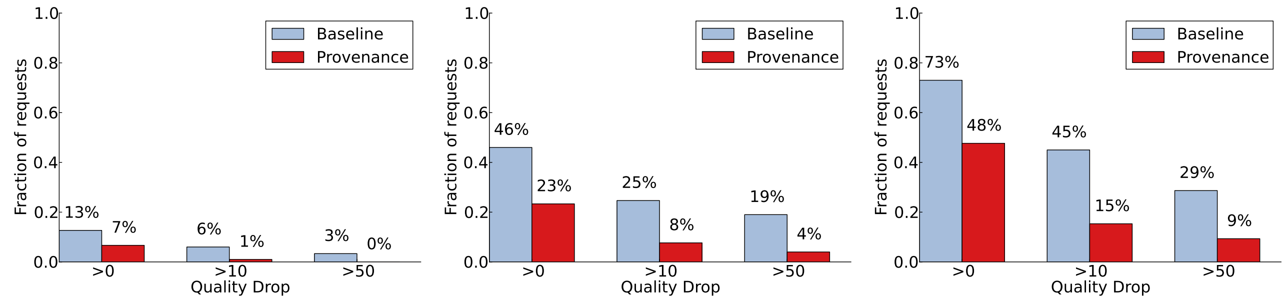
We first used DQBarge to sample production traffic at Facebook and construct a quality model for *Ranker*. DQBarge determined that, by far, the most important provenance parameter affecting quality is the generator used to produce a candidate. For example, one particular generator produces approximately 17% of the top-ranked pages but only 1% of the candidates. Another generator produces only 1% of the top-ranked pages but accounts for 3% of the candidates.

Figure 4 compares the quality of request results for DQBarge with a baseline that makes tradeoffs without using provenance. We sample live Facebook traffic, so the requests in this experiment are different from those used to generate the quality model. We vary the tradeoff rate and measure the *quality drop* of the top ranked page; this is the difference between where the page appears in the request that makes a data-quality tradeoff and where it would appear if no data-quality tradeoffs were made. The ideal quality drop is zero. While Sirius returns a single result, *Ranker* may return up to 3 results. We examined quality drops for the second and third *Ranker* results and found that they are similar to that of the top-ranked result; thus, we only show the top-ranked result for both services.



(a) Tradeoff rate 10% **(b)** Tradeoff rate 50% **(c)** Tradeoff rate 80%

Figure 4: Impact of provenance on *Ranker* quality. We compare response quality using provenance with a baseline that does not consider provenance. Each graph shows the quality drop of the top ranked page, which is the difference between where it appears in the *Ranker* rankings with and without data-quality tradeoffs. A quality drop of 0 is ideal.



(a) Tradeoff rate 10% **(b)** Tradeoff rate 50% **(c)** Tradeoff rate 80%

Figure 5: Impact of provenance on Sirius quality. We compare response quality using provenance with a baseline that does not consider provenance. Each graph shows the quality drop of the Sirius answer, which is the difference between where it appears in the Sirius rankings with and without data-quality tradeoffs. A quality drop of 0 is ideal.

As shown in Figure 4a, at a low tradeoff rate of 10%, using provenance reduces the percentage of requests that experience any quality drop at all from 11% to 6%. With provenance, only 1% of requests experienced a quality drop of more than three, compared to 5% without provenance. Figure 4b shows a higher tradeoff rate of 50%. Using provenance decreases the percentage of requests that experience any quality drop at all from 43% to 33%. Only 3% of requests experienced a quality drop of 10 or more, compared to a baseline result of 17%. Figure 4c compares quality at a high tradeoff rate of 80%. Use of provenance still provides a modest benefit: 59% of requests experience a quality drop, compared to 62% for the baseline. Further, with provenance, the quality drop is 10 or more for only 15% of requests compared with 28% for the baseline.

5.3.2 Sirius

For Sirius, we used k-fold cross validation to separate our benchmark set of questions into training and test data. The training data was used to generate a quality model based on provenance features, which included the language parsing algorithm used, the number of occurrences of key words derived from the question, the length of the data source document considered, and a weighted score relating the query words to the source document.

Figure 5 compares the quality drop for the result returned by Sirius for DQBarge using provenance with a baseline that does not use provenance. As shown in Figure 5a, at a tradeoff rate of 10%, provenance decreases the percentage of requests that see any quality drop at all from 13% to 7%. Only 1% of requests see a quality drop of 10 or more using provenance, compared to 6% for the basline. Figure 5b shows that, for a higher tradeoff rate of 50%, provenance decreases the percentage of requests that see any quality drop from 46% to 23%. Further, only 8% of requests see a quality drop of 10 or more using provenance, compared to 25% for the baseline. Figure 5c shows a tradeoff rate of 80%; provenance decreases the percentage of requests that see any quality drop from 73% to 48%.

5.4 Effect of proactivity

We next examine how proactivity affects data-quality tradeoffs. In this experiment, we send requests to Sirius at a high rate of 120 requests per minute. Without DQBarge, this rate occasionally triggers a 1.5 second timeout for retrieving documents, causing some docu-

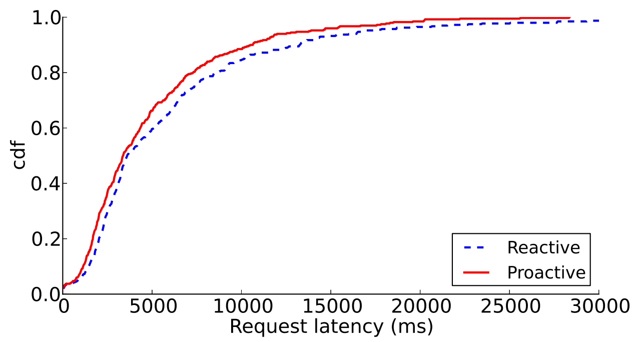


Figure 6: Performance of reactive tradeoffs. This graph compares the distribution of request latencies for Sirius when tradeoffs are made reactively via timeouts and when they are made proactively via DQBarge.

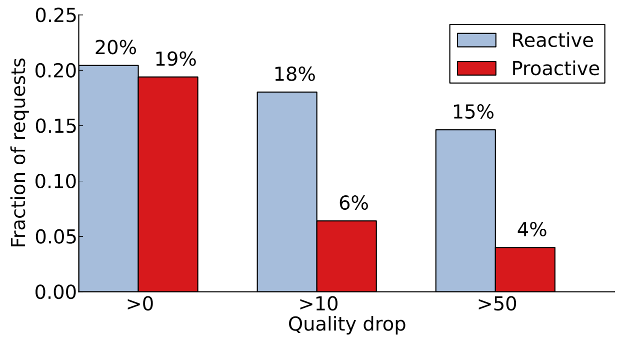


Figure 7: This graph shows that using proactive tradeoffs at a tradeoff rate of 40% can achieve higher quality tradeoffs than using reactive tradeoffs with a timeout of 1.5 s in Sirius.

ments to be left out of the aggregation. These tradeoffs are reactive in that they occur only after a timeout expires. In contrast, with DQBarge, tradeoffs are made proactively at a rate of 40%, a value selected to meet the latency goal of not exceeding the mean latency without DQBarge.

Figure 6 shows request latency as a CDF for both the reactive and proactive methods of making data-quality tradeoffs and Figure 7 shows the quality drop for both methods. The results show that DQBarge proactivity simultaneously improves *both* performance and quality when making tradeoffs. Comparing the two distributions in Figure 6 shows that DQBarge improves performance across the board; e.g., the median request latency is 3.4 seconds for proactive tradeoffs and 3.6 seconds for reactive tradeoffs. For quality, DQBarge proactivity slightly decreases the number of requests that have any quality drop from 20% to 19%. More significantly, it reduces the number of requests that have a quality drop of more than 10 from 18% to 6%.

Under high loads, reactive tradeoffs hurt performance because they waste resources (e.g., trying to retrieve documents that are not used in the aggregation). Further, their impact on quality is greater than with DQBarge because timeouts affect a random sampling of the values returned, whereas proactive tradeoffs omit retrieving those documents that are least likely to impact the reply.

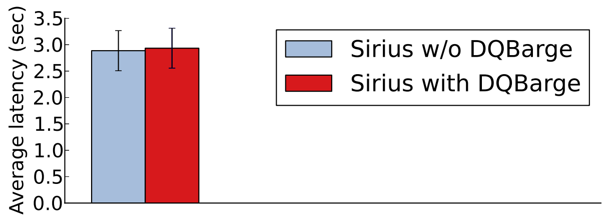


Figure 8: DQBarge Overhead. This graph compares time to process 140 Sirius questions with and without DQBarge; error bars are 95% confidence intervals.

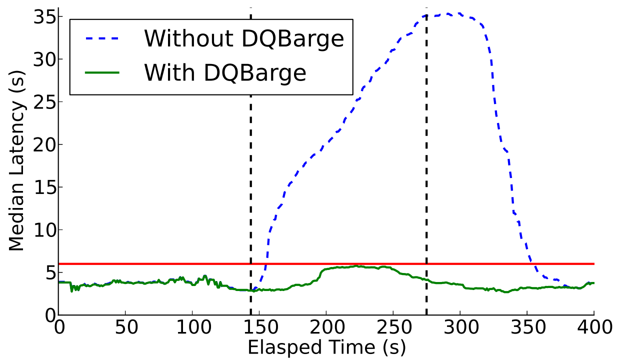


Figure 9: Response to a load spike. DQBarge makes data-quality tradeoffs to meet a median latency goal of 6 seconds.

5.5 Overhead

We measured the online overhead of DQBarge by comparing the mean latency of a set of 140 Sirius requests with and without DQBarge. Figure 8 shows that DQBarge added a 1.6% latency overhead; the difference is within the experimental error of the measurements.

DQBarge incurs additional space overhead in message payloads for propagating load metrics, critical path and slack predictions, and provenance features. For Sirius, DQBarge adds up to 176 bytes per request for data such as load metrics and slack predictions. Tracking provenance appends an extra 32 bytes per object; on average, this added 14% more bytes per provenance-annotated object.

5.6 End-to-end case studies

We next evaluate DQBarge with three end-to-end case studies on our Sirius testbed.

5.6.1 Load spikes

In this scenario, we introduce a load spike to see if DQBarge can maintain end-to-end latency and throughput goals by making data-quality tradeoffs. We set a target median response rate of 6 seconds. Normally, Sirius receives 50 requests/minute, but it experiences a two-minute load spike of 150 requests/minute in the

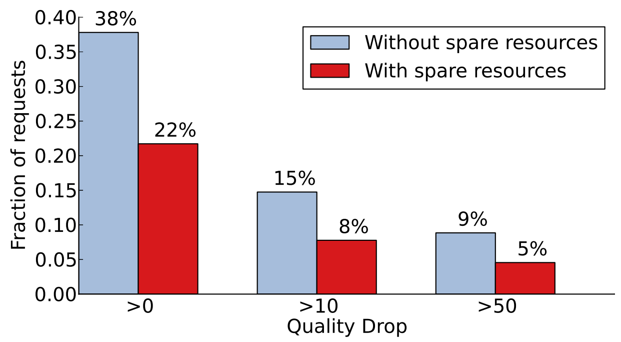


Figure 10: Quality improvement using spare resources. DQBarge uses slack in request pipeline stages to improve response quality.

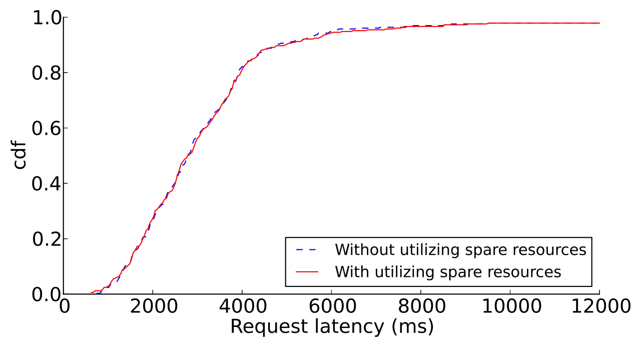


Figure 11: Performance impact of using spare resources. When DQBarge uses slack in request pipeline stages, it does not impact end-to-end latency.

middle of the experiment. Figure 9 shows that without DQBarge, the end-to-end latency increases significantly due to the load spike. The median latency within the load spike region averages 25.2 seconds across 5 trials.

In comparison, DQBarge keeps median request latency below the 6 second goal throughout the experiment. Across 5 runs, the median end-to-end latency during the spike region is 5.4 seconds. In order to meet the desired latency goal, DQBarge generally selects a tradeoff rate of 50%, resulting in a mean quality drop of 6.7.

5.6.2 Utilizing spare resources

Next, DQBarge tries to use spare capacity and slack in the request processing pipeline to increase quality without affecting end-to-end latency. Sirius is configured to use both its default Wikipedia database and the Bing Search API [6] to answer queries. Each source has a separate pipeline that executes in parallel before results from all sources are compared at the end. The Bing pipeline tends to take longer than the default pipeline, so slack typically exists in the default pipeline stages.

As described in Section 4.2, DQBarge predicts the critical path for each request and the slack for pipeline stages not on the critical path. If DQBarge predicts there is slack available for a processing pipeline, it reduces the tradeoff frequency to increase quality until the predicted added latency would exceed the predicted slack. To give DQBarge room to increase quality, we set the default tradeoff rate to 50% for this experiment; note that this simply represents a specific choice between quality and latency made by the operator of the system.

Figure 10 shows that DQBarge increases quality for this experiment by using spare resources; the percentage of requests that exprience any quality drop decreases from 38% to 22% (as compared to a full-fidelity response with no data-quality tradeoffs). Figure 11 shows a CDF of request response times; because the extra processing occurs off the critical path, the end-to-end request latency is unchanged when DQBarge attempts to employ only spare resources to increase quality.

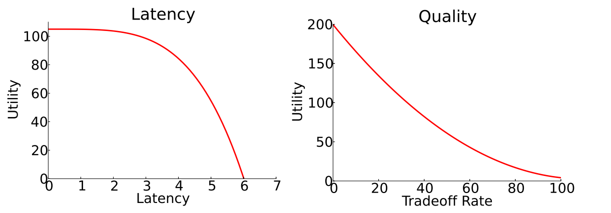


Figure 12: Utility parameters for dynamic capacity planning. These values are added together to calculate final utility.

5.6.3 Dynamic capacity planning

Finally, we show how DQBarge can be used in dynamic capacity planning. We specify a utility function that provides a dollar value for reducing latency, improving quality, and provisioning additional servers. The utility of latency and quality are shown in Figure 12. DQBarge makes data-quality tradeoffs that maximize the utility function at the incoming request rate.

In this scenario, we examine the benefit of using DQBarge to decide when to provision additional resources. We compare DQBarge with dynamic capacity planning against DQBarge without dynamic capacity planning. Figure 13 shows the total utility of the system over time. When the request rate increases to 160 requests per minute, DQBarge reports that provisioning another server would provide a net positive utility. Using this server increases utility by an average of 58% compared to a system without dynamic capacity planning.

DQBarge is also able to reduce the number of servers in use. Figure 13 shows that when the request rate subsides, DQBarge reports that taking away a server maximizes utility. In other words, the request rate is low enough that using only one server maximizes utility.

6 Related work

Although there is an extremely rich history of quality-of-service tradeoffs [7, 25, 29] and approximate computing [4, 8, 19, 18, 28, 30] in software systems, our work focuses specifically on using the causal propagation of request information and data provenance to make better

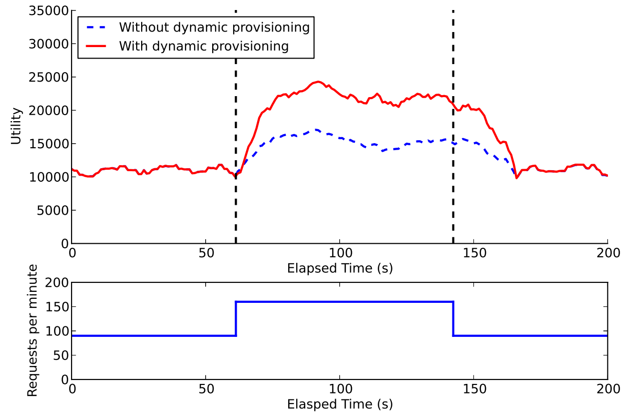


Figure 13: Benefit of dynamic capacity planning. With dynamic capacity planning, DQBarge improves utility by provisioning an additional server. When it is no longer needed, it removes the additional server.

data-quality tradeoffs in low-level software components. Our study revealed the need for such an approach: existing Facebook services make mostly reactive tradeoffs that are suboptimal due to limited information. Our evaluation of DQBarge showed that causal propagation can substantially improve both request performance and response quality.

Many systems have used causal propagation of information through distributed systems to trace related events [5, 9, 10, 13, 23, 26, 27, 31]. For example, Pivot Tracing [23] propagates generic key-value metadata, called baggage, along the causal path of request processing. DQBarge uses a similar approach to propagate specific data such as provenance, critical path predictions, and load metrics.

DQBarge focuses on data-quality tradeoffs in Internet service pipelines. Approximate Query Processing systems trade accuracy for performance during analytic queries over large data sets [1, 2, 3, 17, 22]. These systems use different methods to sample data and return a representative answer within a time bound. BlinkDB [2] uses an error-latency profile to make tradeoffs during query processing. Similarly, ApproxHadoop [14] uses input data sampling, task dropping, and user-defined approximation to sample the number of inputs and bound errors introduced from approximation. These techniques are similar to DQBarge's performance and quality models, and DQBarge could potentially leverage quality data from ApproxHadoop in lieu of generating its own model.

LazyBase [12] is a NoSQL database that supports trading off data freshness for performance in data analytic queries. It is able to provide faster read queries to stale-but-consistent versions of the data by omitting newer updates. It batches and pipelines updates so that intermediate values of data freshness can be queried. Similar to how LazyBase uses data freshness to make a tradeoff, DQBarge uses its quality model to determine the best tradeoff that minimizes the effect on the quality.

Some Internet services have been adapted to provide partial responses after a latency deadline [16, 21, 22]. They rely on timeouts to make tradeoffs, whereas the tradeoffs DQBarge makes are proactive. PowerDial [19] adds knobs to server applications to trade performance for energy. These systems do not employ provenance to make better tradeoffs.

7 Conclusion

In this paper, we showed that data-quality tradeoffs are prevalent in Internet service pipelines through a survey of existing software at Facebook. We found that such tradeoffs are often suboptimal because they are reactive and because they fail to consider global information. DQBarge enables better tradeoffs by propagating data along the causal path of request processing and generating models of performance and quality for potential tradeoffs. Our evaluation shows that this improves responses to load spikes, utilization of spare resources, and dynamic capacity planning.

Acknowledgments

We thank the anonymous reviewers and our shepherd, Rebecca Isaacs, for their thoughtful comments. We also thank Qi Hu and Jason Brewer for their help with the Facebook infrastructure. This work has been supported by the National Science Foundation under grant CNS-1421441. Any opinions, findings, conclusions, or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the National Science Foundation.

References

[1] Swarup Acharya, Phillip B. Gibbons, Viswanath Poosala, and Sridhar Ramaswamy. The aqua approximate query answering system. In *Proceedings of the 1999 ACM SIGMOD International Conference on Management of Data*, 1999.

[2] Sameer Agarwal, Barzan Mozafari, Aurojit Panda, Henry Milner, Samuel Madden, and Ion Stoica. Blinkdb: Queries with bounded errors and bounded response times on very large data. In *Proceedings of the 8th ACM European Conference on Computer Systems*, 2013.

[3] Brian Babcock, Surajit Chaudhuri, and Gautam Das. Dynamic sample selection for approximate query processing. In *Proceedings of the 2003 ACM SIGMOD International Conference on Management of Data*, 2003.

[4] Woongki Baek and Trishul M. Chilimbi. Green: A framework for supporting energy-conscious programming using controlled approximation. In *Proceedings of the ACM SIGPLAN 2010 Conference on Programming Language Design and Implementation*, 2010.

[5] Paul Barham, Austin Donnelly, Rebecca Isaacs, and Richard Mortier. Using Magpie for request extraction and workload modelling. In *Proceedings of the 6th Symposium on Operating Systems Design and Implementation*, pages 259–272, San Francisco, CA, December 2004.

[6] `https://datamarket.azure.com/dataset/bing/search`.

[7] Josep M. Blanquer, Antoni Batchelli, Klaus Schauser, and Rich Wolski. Quorum: Flexible quality of service for internet services. Proceedings of the 2nd USENIX Symposium on Networked Systems Design and Implementation, 2005.

[8] Michael Carbin, Deokhwan Kim, Sasa Misailovic, and Martin C. Rinard. Proving acceptablility properties of relaxed nondeterministic approximate programs. In *Proceedings of the ACM SIGPLAN 2012 Conference on Programming Language Design and Implementation*.

[9] Anupam Chanda, Alan L. Cox, and Willy Zwanepoel. Whodunit: Transactional profiling for multi-tier applications. In *Proceedings of the 2nd ACM European Conference on Computer Systems*, Lisboa, Portugal, March 2007.

[10] Mike Y. Chen, Emre Kiciman, Eugene Fratkin, Armando Fox, and Eric Brewer. Pinpoint: Problem determination in large, dynamic Internet services. In *Proceedings of the 32nd Annual IEEE/IFIP International Conference on Dependable Systems and Networks*, pages 595–604, Bethesda, MD, June 2002.

[11] Michael Chow, David Meisner, Jason Flinn, Daniel Peek, and Thomas F. Wenisch. The Mystery Machine: End-to-end performance analysis of large-scale internet services. In *Proceedings of the 11th Symposium on Operating Systems Design and Implementation*, October 2014.

[12] James Cipar, Greg Ganger, Kimberly Keeton, Charles B. Morrey III, Craig A.N. Soules, and Alistair Veitch. Lazybase: Trading freshness for performance in a scalable database. In *Proceedings of the 7th ACM European Conference on Computer Systems*.

[13] Rodrigo Fonseca, George Porter, Randy H. Katz, Scott Shenker, and Ion Stoica. X-trace: A pervasive network tracing framework. In *Proceedings of the 4th USENIX Symposium on Networked Systems Design and Implementation*, pages 271–284, Cambridge, MA, April 2007.

[14] Inigo Goiri, Ricardo Bianchini, Santosh Nagarakatte, and Thu D. Nguyen. Approxhadoop: Bringing approximations to mapreduce frameworks. In *Proceedings of the 20th International Conference on Architectural Support for Programming Languages and Operating Systems*, Istanbul, Turkey, March 2015.

[15] Johann Hauswald, Michael A. Laurenzano, Yunqi Zhang, Cheng Li, Austin Rovinski, Arjun Khurana, Ron Dreslinski, Trevor Mudge, Vinicius Petrucci, Lingjia Tang, and Jason Mars. Sirius: An open end-to-end voice and vision personal assistant and its implications for future warehouse scale computers. In *Proceedings of the 20th International Conference on Architectural Support for Programming Languages and Operating Systems*, 2015.

[16] Yuxiong He, Sameh Elnikety, James Larus, and Chenyu Yan. Zeta: Scheduling interactive services with partial execution. In *Proceedings of the Third ACM Symposium on Cloud Computing (SOCC '12)*, 2012.

[17] Joseph M. Hellerstein, Peter J. Haas, and Helen J. Wang. Online aggregation. In *Proceedings of the 1997 ACM SIGMOD International Conference on Management of Data*, SIGMOD '97, 1997.

[18] Henry Hoffmann. Jouleguard: Energy guarantees for approximate applications. In *Proceedings of the 25th ACM Symposium on Operating Systems Principles*, 2015.

[19] Henry Hoffmann, Stelios Sidiroglou, Michael Carbin, Sasa Misailovic, Anant Agarwal, and Martin Rinard. Dynamic knobs for responsive power-aware computing. In *Proceedings of the 16th International Conference on Architectural Support for Programming Languages and Operating Systems*, Newport Beach, California, March 2011.

[20] Raj Jain. *The Art of Computer Systems Performance Analysis: Techniques for Experimental Design, Measurement, Simulation, and Modeling*. Wiley, 1991.

[21] Virajith Jalaparti, Peter Bodik, Srikanth Kandula, Ishai Menache, Mikhail Rybalkin, and Chenyu Yan. Speeding up distributed request-response

workflows. In *Proceedings of the Symposium on Communications Architectures and Protocols (SIGCOMM '13)*, 2013.

[22] Gautam Kumar, Ganesh Ananthanarayanan, Sylvia Ratnasamy, and Ion Stoica. Hold 'em or fold 'em? aggregation queries under performance variations. In *Proceedings of the 11th ACM European Conference on Computer Systems*, 2016.

[23] Jonathan Mace, Ryan Roelke, and Rodrigo Fonseca. Pivot tracing: Dynamic causal monitoring for distributed systems. In *Proceedings of the 25th ACM Symposium on Operating Systems Principles*, 2015.

[24] Justin Meza, Dmitri Perelman, Wonho Kim, Sonia Margulis, Daniel Peek, Kaushik Veeraraghavan, and Yee Jiun Song. Kraken: A framework for identifying and alleviating resource utilization bottlenecks in large scale web services. In *Proceedings of the 12th Symposium on Operating Systems Design and Implementation*, Savannah, GA, November 2016.

[25] Brian D. Noble, M. Satyanarayanan, Dushyanth Narayanan, J. Eric Tilton, Jason Flinn, and Kevin R. Walker. Agile application-aware adaptation for mobility. In *Proceedings of the 16th ACM Symposium on Operating Systems Principles*, pages 276–287, Saint-Malo, France, October 1997.

[26] Lenin Ravindranath, Jitendra Padjye, Sharad Agrawal, Ratul Mahajan, Ian Obermiller, and Shahin Shayandeh. AppInsight: Mobile app performance monitoring in the wild. In *Proceedings of the 10th Symposium on Operating Systems Design and Implementation*, Hollywood, CA, October 2012.

[27] Lenin Ravindranath, Jitendra Pahye, Ratul Mahajan, and Hari Balakrishnan. Timecard: Controlling user-perceived delays in server-based mobile applications. In *Proceedings of the 24th ACM Symposium on Operating Systems Principles*, Farmington, PA, October 2013.

[28] Adrian Sampson, Werner Dietl, Emily Fortuna, Danushen Gnanapragasam, Luis Ceze, and Dan Grossman. Enerj: Approximate data types for safe and general low-power consumption. In *Proceedings of the ACM SIGPLAN 2011 Conference on Programming Language Design and Implementation*, 2011.

[29] Kai Shen, Hong Tang, Tao Yang, and Lingkun Chu. Integrated resource management for cluster-based internet services. In *Proceedings of the 5th Symposium on Operating Systems Design and Implementation*, Boston, Massachusetts, December 2002.

[30] Stelios Sidiroglou, Sasa Misailovic, Henry Hoffmann, and Martin Ricard. Managing performance vs accuracy trade-offs with loop perforation. In *Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering*, 2011.

[31] Benjamin H. Sigelman, Luiz Andr Barroso, Mike Burrows, Pat Stephenson, Manoj Plakal, Donald Beaver, Saul Jaspan, and Chandan Shanbhag. Dapper, a large-scale distributed systems tracing infrastructure. Technical report, Google, Inc., 2010.

[32] `http://trec.nist.gov/`.

[33] Ashish Thusoo, Joydeep Sen Sarma, Namit Jain, Zheng Shao, Prasad Chakka, Suresh Anthony, Hao Liu, Pete Wyckoff, and Raghotham Murthy. Hive – a warehousing solution over a map-reduce framework. In *35th International Conference on Very Large Data Bases (VLDB)*, Lyon, France, August 2009.